Vector Based Web Visualization of Geospatial Big Data

An approach to optimize geospatial big data for interactive web based applications
Bsc. Thesis by Florian Zouhar from Darmstadt
Day of exam:

Supervisor: Prof. Dr. Ir. Arjan Kuijper
Advisor: MSc. Ivo Senner

TECHNISCHE
UNIVERSITÄT
DARMSTADT

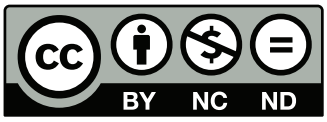
Department of Computer Science
Geospatial Information Management

Vector Based Web Visualization of Geospatial Big Data
An approach to optimize geospatial big data for interactive web based applications

Submitted Bsc. Thesis by Florian Zouhar from Darmstadt

Supervisor: Prof. Dr. Ir. Arjan Kuijper
Advisor: MSc. Ivo Senner

Day of submit:



Thesis Statement pursuant to §22 paragraph 7 and §23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Florian Zouhar, have written the submitted thesis independently pursuant to §22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

In the submitted thesis the written copies and the electronic version for archiving are pursuant to §23 paragraph 7 of APB identical in content.

For a thesis of the Department of Architecture, the submitted electronic version corresponds to the presented model and the submitted architectural plans.

Darmstadt, September 17, 2018

(F. Zouhar)

Abstract

Today, big data is one of the most challenging topics in computer science. To give customers, developers or domain experts an overview of their data, one needs to visualize these. They need to explore their data, using visualization technologies on high level but also in detail. As base technology, visualizations can be used to do more complex data analytic tasks. In case data contains geospatial information it becomes more difficult, because nearly every user has a well trained experience how to explore geographic information. These map applications provide an interface, in which users can zoom and pan over the whole world. This thesis focuses on evaluating one approach to visualize huge sets of geospatial data in modern web browsers. The contribution of this work is, to make it possible to render over one million polygons integrated in a modern web application which is done by using 2D Vector Tiles. Another major challenge is the web application, which provides interaction features like data-driven filtering and styling of vector data for intuitive data exploration. The important point is memory management in modern web browsers and its limitations.

Acknowledgements

At the beginning of the research, the task to start was storing and visualizing huge geospatial datasets in a modern web application. The data-driven bioeconomy project Databio[1] was the starting point for this research. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 732064. It is also part of the Big Data Value Public-Private Partnership[2].

The main task was to visualize agriculture datasets including geometries and additional information to certain agriculture parcels. In addition to that interaction should be implemented like filtering time series or do data-driven colorization of the parcels. The partner in our pilot provided a huge comma-separated value (CSV) file for evaluation of the technical solution.

Out of this, multiple limitations and issues, using state of the art solutions occurred. We investigated many ideas how to solve the problems. It is a very interesting topic and i like to go further in research to improve existing ideas and technologies. There is definitely a gap, which has to be closed in this topic.

Special thanks to my advisor Ivo Senner who introduced me to this project and supported my ideas all the way through the research.

[1] DataBio - https://databio.eu

[2] BDV PPP - http://www.bdva.eu/PPP

Contents

1 Introduction

Today, computer scientists are able to store massive amounts of data of all kinds. Data is used to create statistics and analyse behaviors of machines, humans, the nature and much more. Algorithms from data aggregation to machine learning are applied, to gather more information and have a chance to interpret these massive amounts of data.

Adding geospatial references to data makes it more important. Users, domain experts, developers and scientist can handle such data much better, as humans have an intuition of geographic context on earth. Hence, it is a essential topic to visualize data not only as statistic diagrams or complex interactive graphs. Geospatial data should be visualized on a map to relate to the geographic context. Users can explore the data by paning and zooming the world like they are used to by applications like Google Maps [3] or OpenStreetMap [4]. Users are well trained on using these interfaces to explore countries, streets, cities, buildings or even satellite images.

There are two techniques to provide this map interface. The most common approach used for street maps is to slice the world into tiles, render raster images and send these to the browser. The other option is to load vector data into a browser which has to render these to provide an interactive interface. The main challenge here is to visualize a massive amount of geometries without loosing render performance and therefore interactivity for the user.

State of the art solutions are mostly dealing with static data which makes a visualization much easier. Streets or buildings change only some times a year, so for example Google updates there map application only several times a year. They pre-render the whole world, which is a very time consuming task and therefore, this method cannot be applied on daily changing data.

The existing solution to load vector data to a browser is a Web Feature Service (WFS)[5] implementation. Users explicitly define which part of the world and which type of data they like to explore and trigger the data preparation by hand. Having spatially dense data, distributed over a large area, cannot be explored interactively using this technique. One advanced example is visualizing information about growing crops in millions of parcels spread over the country. The information about the growing plants, trees and grain types are updated every day, which makes the data dynamic. The main challenge is to provide a map interface to explore this amount of dense data using a vector based approach.

The contribution of this work is to evaluate an approach which yields the benefits of both solutions mentioned before. At first make use of vector data only. This is very important to provide interaction features like exploring temporal datasets or apply data-driven stylings without reloading data from servers.

The next important point is, this work provides a well known user interface. Users have to be able to zoom and pan through the world like on any street map. The vector data is sliced into tiles, as done using raster images. These tiles are transformed to a optimized format called Vector Tiles.

This work focusses on evaluating this approach. To do so, it is focused on vector based tiling approaches and suitable javascript frameworks for layer based map applications. It is not about introducing a geo information system for the web or implementing a massive cloud storage solution. Research was also not focussed on finding the one solution, as research in this particular direction is in its early states.

There are three essential steps this solution is providing. At first data has to be stored in a way to have efficient geospatial access. This is achieved by using a geospatial index along with a fast scalable distributed filesystem.

Secondly the data has to be optimized for visualization purposes. This is done by implementing a tiling algorithm and transforming the data to a specialized format. This format has the benefits of faster visualization and faster network transmission while efficiently saving storage space.

At last data has to be transmitted to a web application running in modern web browsers. The geometries then are rendered using a WebGL map application framework. Properties, for example the type of one parcel, are already attached to those geometries. Also it is possible to add interaction concepts like filtering data, user defined styling of geometries or even manipulate, delete and add geometries.

[3] Google Maps - `https://maps.google.com`

[4] OSM - `https://www.openstreetmap.org/`

[5] OGC WFS - `http://www.opengeospatial.org/standards/wfs`

Goals to reach during the research process are described in the next section.

1.1 Research Objectives

The main goal of this research was to evaluate an approach of visualizing huge amounts of geospatial data in modern web browsers. This should be achieved while having the opportunity to interact with data directly. This includes sub goals to reach.

The daily growing geospatial database can be accessed fast and stable. At any point of processing data, the reference to the original data portion has to be available and correct.

A collection of geometries should be lightweight to be easy transferred over a network and provide users the ability to explore it. Data newly added to the main data storage should be visible in a reasonable time, too.

The transmitted geometry collections should be visualized as requested by the user. A user interface should be provided as a web application to interact with the data.

1.2 Method

At the beginning of the research the task to start was storing and visualizing huge geospatial datasets in a modern web application. To do so state of the art solutions were evaluated whether they could provide a suitable solution.

A real huge dataset was provided in context of the DataBio project for evaluation of the technical solution. These databases are growing dynamically with one snapshot every two weeks. Data was provided in comma separated (CSV) files.

To do a structured well defined research the following steps are gone through. General requirements and goals are defined to have the initial idea what the research should achieve. After that, it is necessary to get used to relevant background topics. Then related work and related topics have to be collected in order to define the staring point. Having a decent overview about state of the art techniques and technologies, problems and missing features are written down. The next step is to develop an concept to solve these problems. Additionally suitable technologies are collected to be extended during the implementation phase. Now implementing a prototype and periodically evaluate the progress is the next step. To evaluate the progress one has to keep the research objectives in mind.

After this phase the proof-of-concept prototype is used to evaluate the results and whether the approach found, solves the initial issues and goals. This work focusses on one approach and the evaluation of the implemented solution. It can be seen as a starting point of research in the topic of interactive visualization of geospatial big data. The database is not a static dataset, but data is added and deleted dynamically. Changes have a direct effect to the visualization.

1.3 Chapter Overview

This section gives a brief overview about the chapters and their content.

At first the relevant background section 2 gives a detailed introduction into common concepts and technologies in the 2D visualization and geospatial data context.

After this, the requirements are defined needed to reach the research objectives [1.1].

The related work[4] chapter gives an overview of research already done in this context or in the context of sub-components. The sub-components are the three main parts of the approach, which are the main storage, the optimization process for visualization purposes and the visualization itself.

A concept is introduced afterwards, which describes how the workflow looks like from data to the interactive visualization. The implementation chapter gets in more detail within used technologies and development approaches.

The evaluation chapter focusses on stating out whether the solution solves issues in the context of bigdata visualizations. It describes the quality of the solution and which parts need further research.

Improvement ideas and experiments are briefly described in the future work chapter. This chapter is a bit longer as usual, because this thesis focusses mainly on evaluating approaches in its context.

2 Relevant Background

This chapter gives an introduction to relevant topics regarding the processing and visualizing of geospatial data. One part is a generic overview, and the other gives a deeper view on geospatial related technical details.

2.1 2D Visualization

As briefly described in the introduction this work focuses on optimization of geospatial data to be suitable to modern browser render technologies. In todays time it is important to provide a flexible way to explore big datasets. In order to get rid of operating system and hardware dependencies modern web browsers allow to utilize graphic accelerating units without installing additional software or having native applications. Also there is less need for native applications when not having critical real time applications.

Thinking of 2D visualizations in web browsers leads to the straight forward solution of pre-rendered raster images. Images are split into tiles and stored in a tile tree structure. Mostly this is a simple directory tree and PNG files on a hard disk. The tiling technique is explained in detail later on.

Javascript map libraries like OpenLayers [2.8.1] are providing a user interface to scroll through the world using these tiled raster images. The benefit is requesting, painting and caching is easy and all done by the browser. The downside is the lack of interactivity. As images cannot be manipulated without being re-rendered, geometries can not be manipulated by the user.

The only way to keep maps fully interactive is using vector data directly in the browser. Geometry collections are loaded into memory, transformed into pixel coordinates and rendered. As speaking of browsers and vector data there are two main techniques to render these.

The first one is using HTML Document Object Model (DOM)1 and Scalable Vector Graphics (SVG). SVG is a XML encoded data format and provides a simple way to load vector based data into browsers. They are loaded into the DOM and then rendered by the browsers engine. The SVG container and its members can be manipulated using well known mark up languages like Cascading Style Sheet and HTML. This makes the use of SVG very straight forward because the browsers engine is doing all the rendering work. While adding more data, complexity and more precision browsers are not able to handle such a huge DOM anymore. A DOM having more than a thousand elements is already slowing down the app responsiveness [6].

The second technique is the HTML Canvas using WebGL. Using Javascript and suitable libraries like OpenLayers or Mapbox GL JS browsers make use of the GPU to directly render data onto a canvas. This can be used combining tiling and vector data to efficiently fetch these vectorized tiles from a server and render vector data onto a canvas. Interaction gets more complicated to implement using a canvas instead of SVG but is still full featured possible. Interaction in this context means to do data-driven filtering of geometries, manipulate the geometry itself or even add new user defined geometries.

2.2 Tiling

Geospatial map applications are giving users exploration and browsing capabilities. It is mostly possible to pan and zoom over the whole world. Loading one huge world image and just zoom and pan on it would lead to either bad image qualities on higher zoom levels or allocating to much memory and taking huge amounts of transmission time when using high resolution images. To avoid both problems the world is sliced into tiles. This goes from top to bottom. Technically the tile tree start with zoom level zero, which includes every geometry. Zoom level 1 has four tiles and every higher zoom level doubles the amount of tiles. Now the browser has to load only relevant tiles for the current viewport and the current level of detail. In practice the starting point in the user interface is mostly zoom level 2. Zoom level 0 and 1 can be shown as well but they are showing the whole earth multiple times on modern computer displays.

In standard map application these tiles are simply raster images loaded from a server. This makes browsing very easy because on fast zooming, requests for tiles can be canceled while getting out of the viewport. Caching is totally up to the browser which is very efficient on images.

As mentioned above, raster images are efficient but no option since interaction within the geometries is limited to selecting these to get additional information. Furthermore, manipulation of the images on demand is very expensive. To do so every tile will be re-rendered on a server and then transmitted over the internet. When thousands of geometries are included in the requested tiles, rendering cannot be done in a reasonable time. But this is actually necessary to change

[6] https://codeburst.io/taming-huge-collections-of-dom-nodes-bebafdba332

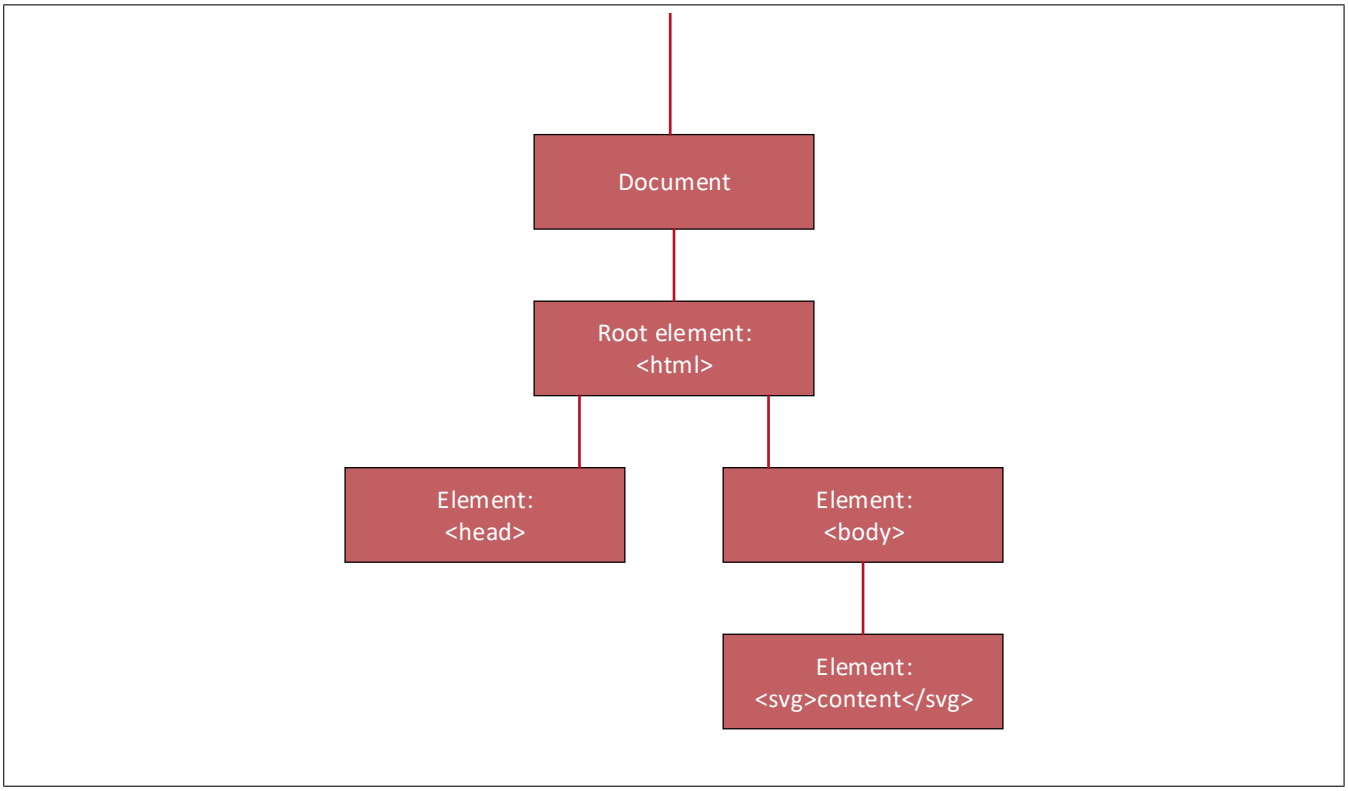


Figure 1: Root of DOM tree

the subset of data to visualize or to apply user defined stylings.

The next section describes how this technique can be adapted a vector based approach.

2.3 Vector Tiles

Interaction with geospatial data needs vector data to work full featured. As described in the last section, tiling is the state of the art standard to provide user interfaces to explore geospatial data. The idea is to process the data exactly like rendering raster images but slicing the vectors itself and store these as vector tiles.

GeoJSON[7] is a open standard used for most web applications when working with geospatial data. It is encoded in Javascript Object Notation syntax and therefore directly readable by modern browsers. Handling GeoJSON files is also easy because it is human readable. Tiling can be applied on a feature collection and each tile can be stored in the GeoJSON representation.

This is an approach to at least be able to explore data using a map application and vector data directly. For example PostGIS has actually implemented a interface to export tiles in GeoJSON format.

2.3.1 Google Protobuf

Google Protocol Buffers[8] (Protobuf) is a mechanism for serializing structured data. There are available implementations for all popular programming languages. At first one defines a scheme readable by a Protobuf compiler. Using Java, the result would be a Class definition which can be used to represent ones structured data. On transmitting an object, it is serialized, transmitted over a network and can be re-constructed based on the scheme definition introduced at the beginning.

These technology is used by Mapbox to encode, transmit and store Vector Tiles. They provide an open source specification for it, which is described in the next section.

[7] GeoJSON - `http://geojson.org`
[8] Google Protobuf - `https://developers.google.com/protocol-buffers/`

2.3.2 Mapbox Vector Tiles

GeoJSON has a lot of character overhead by design as it should be human readable. This means there are many syntactic characters, which are curly braces, quotes. Additionally information attached to the geometries is added as a key value store to every feature. Though, assuming every geometry has the same types of properties, every key is stored for every feature. A feature collection containing one thousand features has to store for example the string "address" one thousand times. Mapbox[9] developed a standard called Mapbox Vector Tiles which provides an efficient way to store vector tiles. Geometries are projected into a virtual extend using only integer values. Additional information of these geometries, called properties in GeoJSON are converted to a set of tags. Based on Google Protobuf all features are serialized into a binary format. To improve transmission efficiency even further it is possible to compress the resulting data using gzip.

On a first look this approach is very efficient in saving disk capacities and network bandwidth. More important it allows efficient rendering as all geometry coordinates are already integer values. Computer displays also have only pixels with integer coordinates, which makes drawing lines containing integer vectors easy. The only thing to do is, scale the coordinates to the correct viewport size and add the offset for the correct position. Figure [2] illustrates one tile with an extend of 4096. Then the tile is shifted to its correct position using a offset. Then it can be rendered into correct position within the tile puzzle.

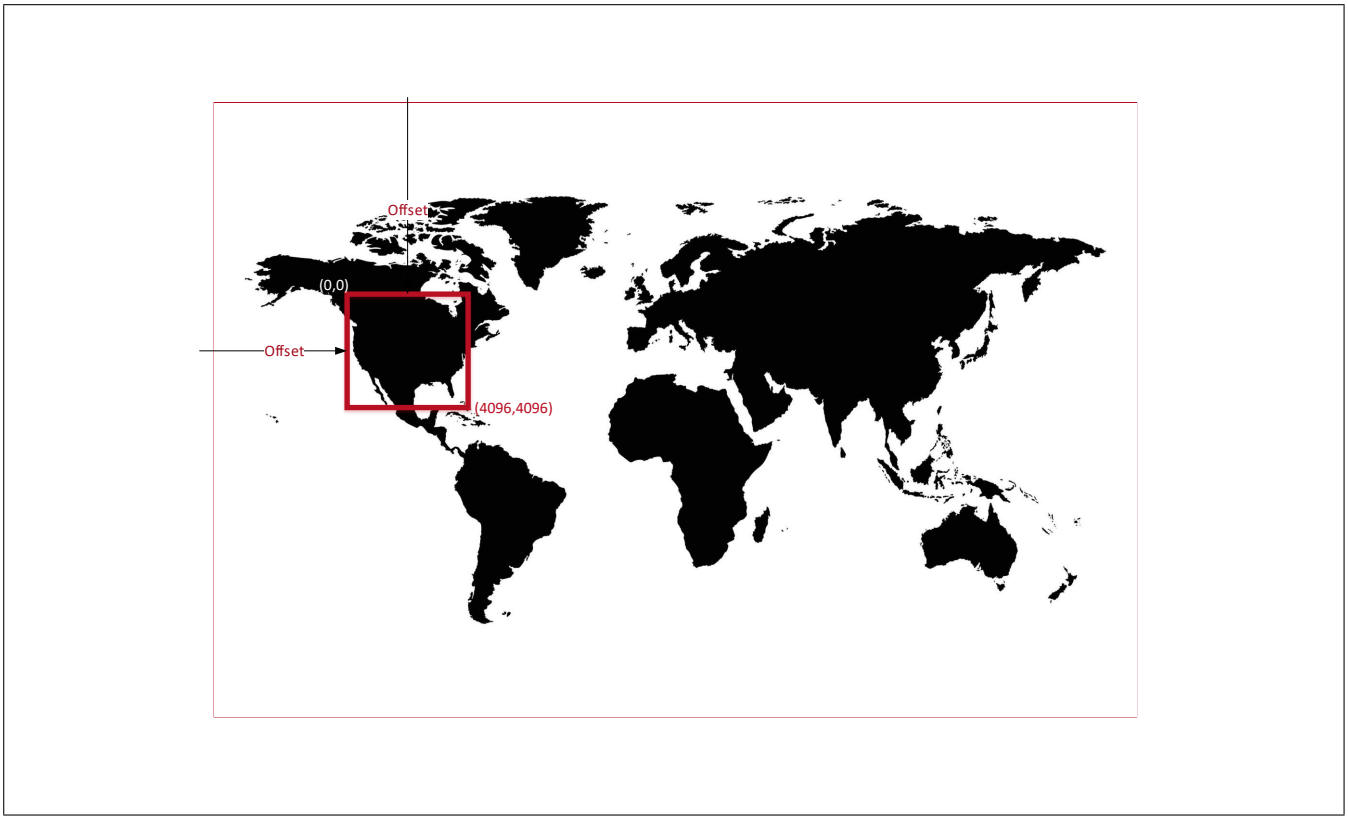


Figure 2: Integer extend of one tile

There is one official Mapbox CLI tool called Tippcannoe[10] to create these tiles. It is written in C++ and reads huge GeoJson files. Tippcannoe provides several features and parameters to adjust the tile creation process.

The tiling process begins with reading all features from the file to memory. Since there is no buffer mechanisms, it is not possible to process data exceeding the available working memory
Before the actual first tile is created, all coordinates are projected to values between zero and one. There could be geometries exceeding the limits of 180 or -180 degrees. When the tiles are created all geometries are clipped at its borders. Clipping means to actually cut all geometries at the tile borders and is illustrated in figure 7. Geometries overlapping the 180 degree meridian would then loose some part. To avoid that, the part overlapping the 180 or -180 degrees meridian is then given an offset of 360 or -360 in order to move it into the correct coordinate range. This process is called wrapping and is illustrated in figure 6

[9] Mapbox - `https://mapbox.com`
[10] Tippecanoe - `https://github.com/mapbox/tippecanoe`

Once the data is prepared the recursive tiling process starts. The first tile (0/0/0) represents the whole world and therefore includes all geometries. One tile has a fixed extend size which is 4096 by default. This size defines the precision of the tiling database and therefore how much details of the geometries a visible in the end. The tile then has an coordinate system vom zero to 4096 in each direction. The Origin is top left. Now the tile creation Process starts. All geometries having coordinates from zero to one must be transformed to the tile extend (0 - 4096) and converted to the Mapbox Vector Tile scheme defined in Google Protobuf.

To keep the integer values small, coordinates are converted to vectors building a polyline. A polyline is defined by a sequence of vectors. First the starting point is given relative $(0,0)$, for example $(1,2)$. The next point coordinates are defined by a vector from the previous point, for example $(3,-1)$. This tile now is encoded binary using Google Protobuf and finally compressed with gzip. The first tile is now ready to be stored.

The resulting tiles are stored using a simple SQLite database with four fields. Three according to the tile number or coordinate (z,x,y) and one for the actual binary tile. This is the default database Mapbox is using for storing tiles.

Each stored tile splits into four new tiles of same size on the next zoom level. All geometries are clipped at the according bounding box of these four tiles and are created as described in the last paragraph.

On calling this recursive algorithm the tile tree is built. Using a simple SQLite client the tiles can be accessed and rendered onto a canvas as described above.

The Tippcannoe implementation from Mapbox works on middle sized datasets well, but this process has its limitations. The most important is the size each tile is not allowed to exceed. To reach this goal mapbox reduces the complexity of the imported geometries. The point density will be decreased by removing points and even whole geometries got thrown out of the tile. This leads to information lack and does not give new information regarding the dataset.

One can set parameters to avoid all these simplifications to have tiles as huge as necessary. When all properties available for the geometries are attached to the tiles and geometries are not simplified, the application allocates to much memory and the process is killed by the browser. See the evalutation section 7 for more detail. Some ideas how this very important issue can be solved are explained in several future work sections [**??**, 9.6, 9.7].

2.4 Geotools

When working with programming languages running in a Java Virtual Machine (JVM) and geospatial data one needs models for object oriented programming representing geospatial features. Also standard spatial operations such as transformation of the coordinate reference system are needed. GeoTools[11] is the most popular Java library for this purpose. GeoTools provides a huge set of features and is based on the Java Topology Suite (JTS) [12]. JTS implements an object model for geospatial data according to the OGC Simple Feature Access specification[13].

The Open Geospatial Consortium (OGC) is an international not for profit organization committed to making quality open standards for the global geospatial community [14].

JTS and GeoTools is mostly used to build more specialized tools, libraries and frameworks.

2.5 Indexing

As described in the research objective section one goal is to have easy and fast access to the whole dataset using spatial queries and performing data analytic tasks. This section shortly introduces and explains the manner of indices according to this work.

Data indices are used to provide access in a efficient way, to do specialized queries and complex analytic tasks on. This enables to do fast requests on a subset of data as needed.

[11] GeoTools - `http://geotools.org`
[12] JTS - `https://projects.eclipse.org/projects/locationtech.jts`
[13] OGC Simple Feature Access - `http://www.opengeospatial.org/standards/sfa`
[14] OGC - `http://www.opengeospatial.org/`

This work makes use of it by indexing the geometries itself and attached information which can be numerical values, timestamp or simply text based information. Indices are implemented in relational databases like mySQL and ProstgreSQL, but also in noSQL solutions like MongoDB. Search engine implementations make even more use of it.

geospatial data can be indexed using several algorithms and datastructures. The GeoHash algorithm is one straight forward solution by hashing locations to a unique string. More complex datastructures like the QuadTree or R-Tree are used to store not only points but whole polygons.

2.5.1 Lucene

Apache Lucene[15] is a open source java-based indexing and search technology. It features many query types as full-text, range and field searching. It also provides an aggregation API and multiple datastructures to index spatial data. Lucene is not specialized on any data format, because it does not provide a data storage solution itself. Lucene is only an indexing service.

2.5.2 Elasticsearch

Elasticsearch[16] is an open source search engine based on Apache Lucene. It is able to index and store all kinds of JSON documents and is fully accessible via a REST API. Elasticsearch knows two geospatial data types, the GeoPoint and the GeoShape. Both index geospatial data using the GeoHash algorithm or a QuadTree datastructure.

The most simple one is the GeoPoint. GeoPoint fields are indexed using the GeoHash algorithm. These can be used to find geometries within a given bounding box or within a given distance of a central point. Furthermore, it is possible to aggregate these points into weighted points using a geospatial grid aggregation. This can be used for instance to visualize a huge amount of point as a heat map.

The more complex one is the GeoShape which allows to index geo shapes like boxes and polygons. These field can be queried using bounding boxes or points within a given radius. GeoShapes can be indexed using GeoHashes or QuadTrees. An issue using GeoShapes is, they have no aggregation features.

2.6 Storage

This section focuses on existing storage technologies for geospatial data and binary encoded vector tiles. It is important to have a good overview, because every storage has its benefits and issues. Criteria are not only performance but also maintainability and the way developer can extend or adapt existing features.

2.6.1 PostGIS

Relational databases are the most spread and traditional way to store structured data besides a plain file system. Nearly every relational database uses SQL to maintain and access their data. PostGIS[17] is a feature rich and complex extension for PostgreSQL Databases to store and index geospatial data.

There are many functions in SQL syntax to query, aggregate and convert data which is stored in such a database. The Mapbox Vector Tile format is a supported export format as well. The main limitations on this technology is its architecure which is stable but very complex and a developer have to glue everything together in one monolithic software. Extensions have to be written especially for PostgreSQL. These are written using a combination of SQL statements, C code snippets and configuration files. To do so one has to get into the complex code of PostgreSQL and PostGIS. This makes maintaining a software based on PostgreSQL hard to manage.

The second option along with extending PostGIS is to query data before working with it outside of PostGIS, but this kind of solutions are lacking of performance. A PostgreSQL database, which should handle bigdata fast, needs a lot of hardware resources as well. Importing a complex street map database on a machine having about 16GB on working memory took about one week to build the index. It has a well maintained codebase, but it does not suite well into modern cloud processing and storage solutions. It is a non-distributed database and hard to setup.

Accessing the Vector Tile interface on PostGIS provides not options to configure how to the vector tiles should be created. Every tile is created on demand, which lacks of performance on thousands of polygons per tile.

[15] Lucene - `https://lucene.apache.org/core/`
[16] Elastichsearch - `https://www.elastic.co/products/elasticsearch`
[17] PostGIS - `https://postgis.net`

2.6.2 SQLite

SQlite[18] is a relational, file-based database. It is very straight forward to use. In this topic is important to evaluate this technology because it is the standard storage used by Mapbox to store vector tiles. The performance tests of tiling in the evaluation sections 7 are done using SQLite. Mapbox is using standard SQLite database files with an '.mbtiles' file extension.

2.6.3 MongoDB

MongoDB[19] is a database for JSON formated documents and classified as noSQL database. Instead of having a fixed structure and scheme like traditional SQL databases it is completely up to the user what kind of data it stores.

MongoDB has a well integrated extension called GridFS. It enables the possibility to store binary data into a MongoDB collection.

This could be an alternative to SQLite as it allows to store vector tiles in a distributed environment. As MongoDB itself has less control and data integrity features then an SQL database transaction are much faster. Updating data is faster because updating binary fields using GridFS is not possible. Instead a newer revision of that field is inserted. GridFS uses versioning along with timestamps. A simple garbage collection implementation can solve the problem of an growing database.

2.6.4 H2

H2[20] is a file-based database. It can be used as key-value file database and usage is very straight forward. Because it is file based there is no setup needed. The native Java client is implemented in a non-blocking way. A non-blocking client is very useful to store vector tiles. Tiles created and ready to be stored, are passed to the H2 client, which stores the tile by itself. As this process is non-blocking, the tiling process can proceed before tiles are actually written onto hard disk.

2.7 GeoRocket

GeoRocket[21] is a high-performance data store for geospatial files.

It combines an indexing technology along with a storage solution to a reactive data store for spatial information. On importing a file it splits up data into geospatial features and stores these chunks into the storage and indexes the geometries and attached information using Elasticsearch. Geometries are indexed using Elasticsearch's GeoShape data type and a QuadTree data structure. Natively it supports GeoJSON and CityGML file formats but could be easily extended to new formats and specialized indexing schemes.

Two types of queries using a domain specific language are possible. Search queries filter data by field values or geospatial boundaries. This merges features together to a collection using the original data out of the storage. Aggregation queries are available to perform more complex data analytic tasks. The response of the aggregation requests are JSON formatted. GeoRocket can be used as base storage to build an application onto. The architecture of georocket is illustrated in figure 3.

2.8 Rendering

The most essential part to do is render the actual vectors into the browsers canvas. This section introduces frameworks providing a rich platform to build web based map applications. All these frameworks have in common there architecure. They organize attached sources and map these to one different layer each, which are rendered in exactly this order on top of each other. For example one attaches an OpenStreetMap raster image source and a GeoJSON file as a second source. Then these are added to the map as layers with a given styling. The order being added defines the visible order of the layers. The OpenStreetMap layer is the base and the secondly added GeoJSON file is the rendered on top of these raster images.

2.8.1 OpenLayers

OpenLayers[22] is a feature rich open source JavaScript library to render spatial data into a canvas HTML element. This can be done using the software based renderer. Depending on the operating system, hardware and browser version it

[18] SQLite - `https://www.sqlite.org/index.html`
[19] MongoDB - `https://mongodb.com`
[20] H2DB - `http://www.h2database.com`
[21] GeoRocket - `https://georocket.io`
[22] OpenLayers - `https://openlayers.org`

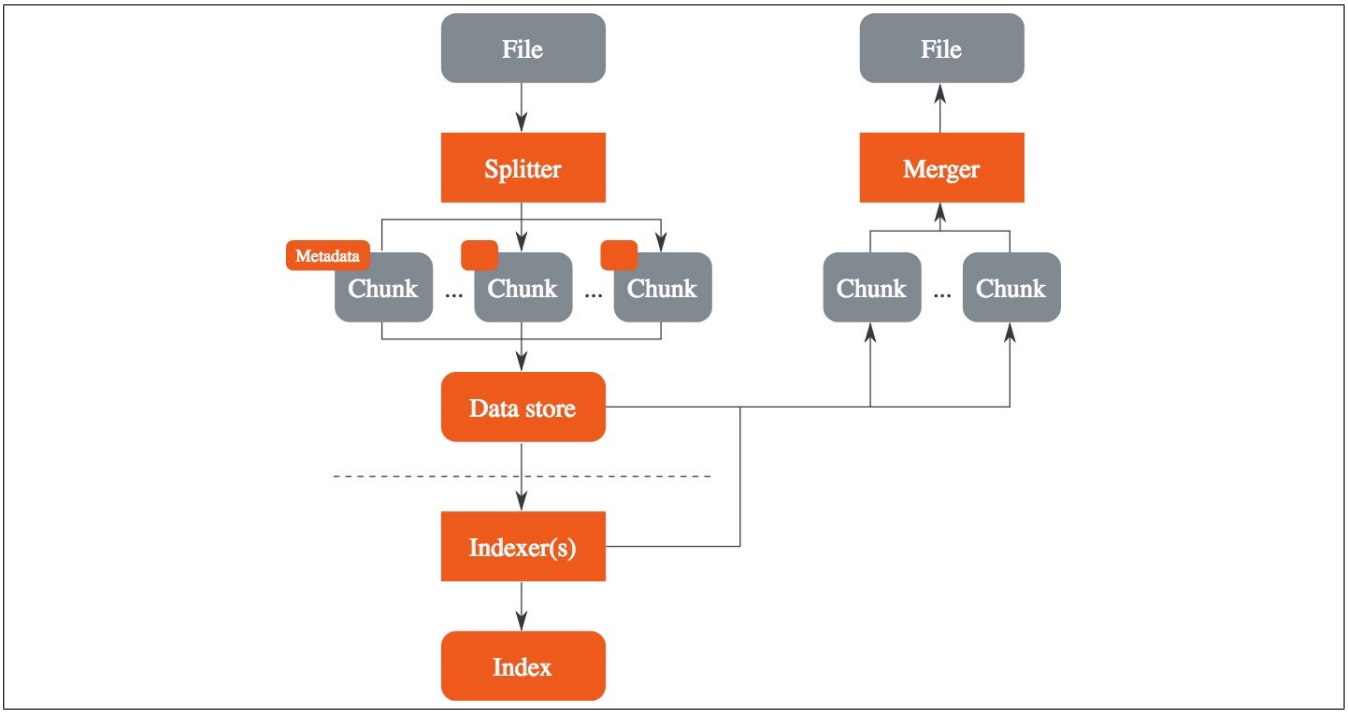


Figure 3: GeoRocket architecture - Source: `https://georocket.io`

uses the WebGL API to make use of the graphic accelerating unit.

OpenLayers supports several formats including GeoJson Feature Collections, WMS servers, TMS servers and Mapbox Vector Tiles. This framework provides features to filter, colorize or interact with features.

This implementation of Mapbox Vector Tiles is integrated as a stable feature in the current release. This feature is still in a early development state. As of this for instance caching is not as efficient as it should be. Also filtering is not implemented for Mapbox Vector Tiles.

2.8.2 Mapbox GL JS

As mentioned in the relevant background section mapbox defined a vector tile standard. To render GeoJSON and Protobuf (PBF) tiles they also implemented a javascript library called Mapbox GL JS[23] which runs using WebGL.

Features implemented for Vector Tiles are filtering features, data-driven colorization and user interaction with certain features.

[23] Mapbox GL JS - `https://github.com/mapbox/mapbox-gl-js`

3 Requirements

This chapter describes requirements to reach the predefined goals. The requirements are mainly resulting from issues and missing capabilities found in state of the art solutions.

3.1 Functional Requirements

The main idea at the starting point was to make huge geospatial agriculture datasets available to be explored and used by agriculture domain users. The following functionalities are required to build a data workflow and user interface suitable to the needs.

- Import huge datasets into a storage technology
- Monitor the importing process
- Access the data efficiently
- Explore geospatial data using a visualization
- Interact within a visualization and do data-driven filtering and styling

These are the functions the user or customer does recognize. Additionally there are some functionalities on the more technical side. The importing of data into the main storage should automatically start the data optimization for visualization purpose. In a reasonable amount of time users should see updated data being visualized.

3.2 Non-Functional Requirements

The following sections describes the non-functional requirements one has in order to work with geospatial big data.

3.2.1 Performance

Performance has always been a topic while handling data and processing data. Spatial data on earth is represented by its coordinates which themselves are stored as floating point numbers. Double precision floating point numbers are used most commonly to be able to render geometries more precise. Using this knowledge every operation on geospatial geometries costs an non irrelevant amount of computation power and time.

In order to successfully build a user interface, responsive enough to have a good user experience, performance is a significant factor. Storing and accessing given datasets is the first step. In order to have the same coordinate reference system (CRS) data is first transformed to one common coordinate system (for example WGS 84).

After that they are imported into a storage solution and indexed in an efficient way. The process of indexing coordinates are commonly done directly on floating point coordinates which needs more computation time that integer pixel operations. Using a common CRS using floating point coordinates to visualize geometries, leads to even more floating point operations. The coordinates are transformed into pixel coordinates every time they are drawn onto the canvas. It is important to do these coordinate transformations and indexing processes fast in order the have user access to the index data and provide a low latency visualization. Doing computations fast is not the only solution, but keeping the number of floating point operation processes low can speed up the whole process, too.

In order to connect the web visualization to a backend storage solution data is usually transmitted over a network. Of course performance of the network itself is important, too, but in most cases this cannot be improved. It is more important to find an efficient data format to transmit a feature collection to frontend side. Hence, performance can be considered important in the manner of data formats in this use case.

The trade-off between the computation time to perform transformations to achieve smaller chunk portions transmitted and the transmission time itself plays an important role in the storing and visualization workflow.

Secondly one should consider how much working memory is available in modern web browsers to keep a lot of geometry properties in memory. Trade offs between fetching additional data fields on demand and having them in memory by default is important, too. In order to do data-driven colorization of the geometries, at least one numeric property is necessary. Every agriculture parcel has for example a water index property attached. When this property is not transmitted to the web application by default, all tiles have to be reloaded when a user applies a custom data-driven styling.

Performance is the most important and the most interesting topic for this research work. Every step processing the data has to be fast enough to reach the goal of providing a user interface, which satisfies the users experience. A web application has to appear very responsive to the users.

3.2.2 Interoperability

Traditional applications are mostly designed for one special purpose and only for special devices and architectures. In the beginning of personal computers desktop applications were built. These were able to run on a general purpose operating system. But even this was limited by the hardware architectures these operating systems are running on.

The aim is to have one general purpose application to access and browse through data or even to do data analytic tasks on datasets. To provide this as a web application allows interoperability. Every state of the art device should be able to access the such an application through a browser.

This requirement is going along with the performance requirement as one has to deal with the trade-off having a cross-platform application not specialized for one operating system and its lack of performance optimizations available in modern web browsers. On the other hand web applications are easier to maintain, update and develope than directly on the operating system installed tools.

In todays digital world security and safety are playing a very important role. Native Applications have to be installed directly on the operating system using administrative rights. Granting administrative right to anyone can result in installing unwanted software which is an security issue.

Also users want to have access to there web application, and therefore to their data from everywere. Also sharing of data or granting access to a web application is much easier then sharing data directly, for example among costumers.

3.2.3 Interaction

Web applications have the huge benefit that a wide variety of user interaction can be implemented. This work focusses on data-driven visualization and interaction with geospatial data. In this particular use case interaction means filtering by geometry properties, data-driven on-demand colorization of geometries and getting more detailed information about features. Furthermore more complex data analytic tasks should be performed which makes references to the original features necessary. This is explained in more detail in the data integrity subsection.

Performing the actual interaction, for example a mouse click, by a user should invoke a change of the visualization directly. The meaning of 'fully interactive' includes that users can see an effect in a certain amount of time. To have a good user experience the response time of the user interface has to be less then a second. This speed of responsiveness one would appreciate to have. Response times over one second are considered as waiting time by users [11].

This work focusses the aspect implementing interaction features. That makes this requirement important. Besides transmitting a lot of data into web browsers it is important to handle it and let users interact with it.

3.2.4 Data Integrity

In most cases spatial information is served in data files, for example CSV or GeoJSON, by customers. For exploring and visualization purposes data is transformed into specialized data formats and data structures. Information gets lost or more information is added to the data as needed. As customers still want to export their original data, integrity is important.

One solution is to keep the original format in a main storage and optimized data is a additional datastore. This concept keeps references to the original data store. This makes it possible to find the original data portion belonging to a optimized geometry used for visualization purposes.

3.2.5 Scalability

There are different aspects of scalability which can be adressed. An application may be able to be used by thousands of users, or even thousands of users entering the platform at once. A processing server provides enough power to process one megabyte but also one terabyte in a short amount of time. In this approach there are two types of scalability to look at.

The optimization processing for visualization purposes should be able to handle many small files but also very huge datasets bigger than one gigabyte of plain text data. The main focus is not processing lightning fast, but processing and storing vector tiles should be finished in a reasonable amount of time.

When it comes to the actual web application fetching tiles and rendering geometries, the application should be able to render a huge amount of polygons. About two million non complex polygon having about six points should be possible to visualize. It is required to hold properties in memory along with the actual geometries itself in order to provide interaction features.

Having an overview over the approach in this work, scalability is a requirement to proof the concept already explained. The workflow concept defines three stages. Importing the original files, optimizing data for visualization purposes and finally provide a responsive user interface to explore the data. The concept should work for small datasets, but also when adding more data over time.

3.2.6 Summary

This section sums up which non-functional characteristics are important and are evaluated in the evaluation section 7.

1. Performance
    - Reduced floating point operations
    - Reduced working memory allocation on backend- and frontend-side
2. Interoperability
    - Operating system independence
    - An application running in modern web browsers
3. Interaction
    - Common interaction features can be implemented
    - Responsiveness as expected by users
4. Data Integrity
    - Referencing to original data is possible at any time
5. Scalability
    - Optimizing data runs in a reasonable amount of time
    - The concept works while importing more data to the system

4 Related Work

There are few researches done already on visualizations, using vector based tiling approaches. This section gives an overview on existing methods in storing, processing and visualizing data in a geospatial context.

In 2014 Vitolo [14] has written a survey on how bigdata is handled using web technologies. In evolution of computing science data collections expend very fast. New technologies are needed in order to handle such a huge amount of data. He describes this has to be possible in manner of analysis, workflows and interaction within this datasets. The paper discusses the current web technologies used to process simple dataset, which can be used very straight forward. In addition to that it get more complicated to handle more complex data in a flexible while keeping the connection to standards maintained by the OGC.

PostGIS is actually the most used storage technology for geospatial data. There exist many solutions based on modern technologies but the ease of use and the lack of geospatial computer scientists make it very common to stick to the plain old and stable PostGIS database. Also often essential usage of OGC standard data formats makes it more easy to implement applications in PostGIS.

End users mostly interact with these databases using dekstop GIS applications. To the best of our knowledge, there is not much work done on providing web based applications able to handle bigdata and provides a platform independent access to customers and end users.

While Vitolo was focused on bigdata and web technologies Yue [16] discusses big data in the manner of geospatial information systems. In his opinion it is important to support the geospatial domain as especially GIS software should be able to handle huge datasets containing geospatial data.

Figure 4 by Yue shows the historical development of the wording bigdata in a geospatial context.

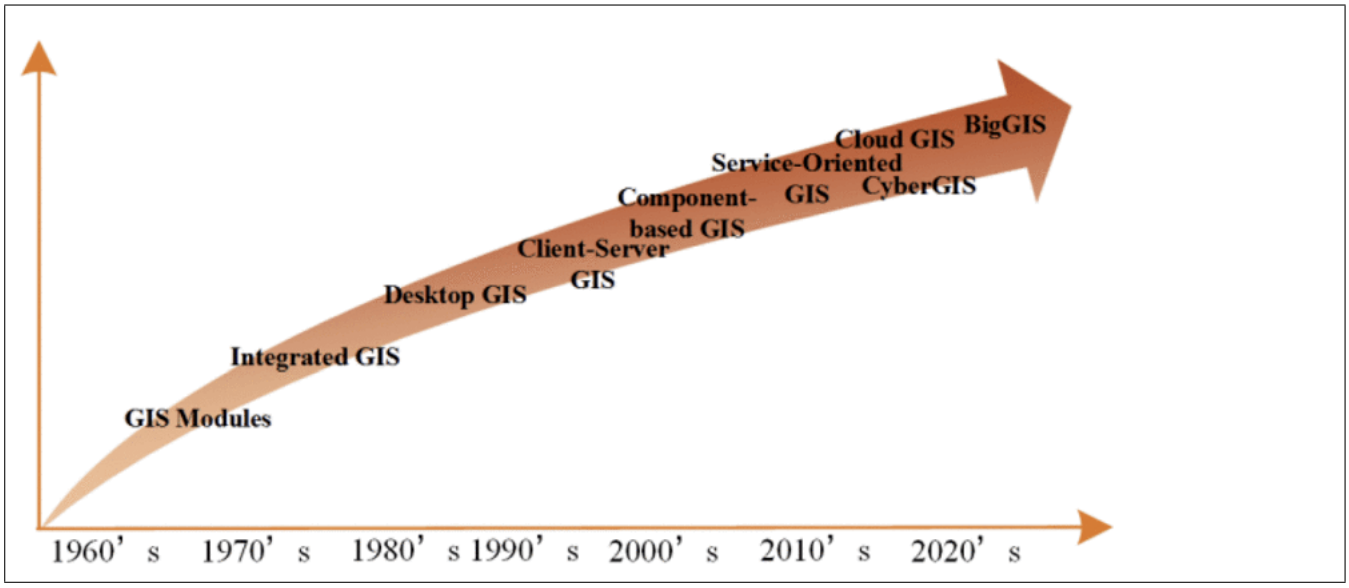


Figure 4: History of Bigdata in a geospatial context, source: Yue [16]

Concepts from Horak [6] in Web Tools for Geospatial Data ManagementThese focus on providing remote interfaces based on XML and defined also by the OGC. But it is not really focused on the bigdata manner and the aspect of exploring the data using a visualization.

As this thesis is about optimizing data for visualization purposes data formats are a important topic to look at. Data format research is mostly done to do faster transmission of spatial data over the network. One approach by Yang und Li [15] is data compression by clustering data. The ideas are to compress data, but not for a visualization purpose.

The concept of tiling is very old also still state of the art. This technology has to be adapted to upcoming requirements. Currently tiles are stored using pre-rendered raster images. While talking about tiling, caching tiles is a widely mentioned topic [10]. Not only in transmitting raster images and do caching on server side, but also caching vector tiles, for example geoJSON vector tiles not optimized for transmitting [1]. Also Blower [2] is already talking abount caching

of tiles and GIS in the Web. Ingensand [7] and Van de Brink [13], they all do discussion on how to get forward with actual existing technologies.

Mostly the approaches focus on one topic, not on combining existing ideas. Straight forward queries fetching data on the backend side and compressing a files for transmission is one approach. But it does not help to provide the ability of scrolling through the data. It is necessary to explicitly query a subset of data from servers.

Ideas of tiling to provide a map user interface are really focussing on tiling but not on trying to compress, cache tiles in a vector format without increasing memory usage at any point in the workflow.

The only stable data format for 2D vector tiling currently is mapbox vector tiles. Eriksson [3] has done review on the map render framework implemented by MapBox called mapbox GL JS. It focuses on a huge amount of features and the performance of the render engine implemented by MapBox. The tile generation is done by using a CLI tool after exporting a GeoJSON file from PostGIS.

It is not looking forward to find a solution on scaling databases but focused on more static street maps imported to PostGIS.

These Vector Tiles introduces by Mapbox are using Google Protobuf. This is an efficient way to serialize and transmit structured data over a network. Feng [5] describes usage in manner of online games. But there is no research besides the vector tile specification using protobuf in the manner of geospatial data. One should go further and evaluate if there are better solutions in order to be more flexible in working with vector tiles and dynamic data.

SVG has to be mentioned here as a part of the development of spatial data visualized in browsers. Visualization ideas used SVG a lot some time ago [9]. But as explained before a canvas is way more efficient while rendering more geometries and has replaced the svg technology in this manner mostly.

Another important topic is server-side rendering. For this these is not very important as interaction is one of the main requirements. But if it is possible to render in real time, and even transmit in real time server-side rendering will get important in this manner.

Olasz [12] has written a survey on the possibilities on using server-side rendering. In particular he uses GeoTrellis[24] and several different storage solutions to evaluate in which manner these technologies can be used. Visualization in the manner of bigdata here is more a small topic. Just creating a tiling tree with raster images is mentioned.

As described for now the state of the art at the moment is to render tiles on servers and accept the lack of interaction possibilities.

To sum this up, there is a lot work done some years ago focussing on data standards and storing data well using state of the art technologies. Also workflows from data towards visualizations are evaluated.

But is is important to evaluate each component not only itself but in the whole processing chain. As there is not standard to to efficient transmission and visualization of geospatial data, experiments have to be done on how to combine and define technologies in order to work together efficient and provide users an intuitive way on exploring data.

The next sections are discussing the concept of doing this research and evaluation.

[24] GeoTrellis - `https://geotrellis.io`

5 Concept

In general this approach evaluates the possibilities to visualize a huge amount of data in modern web browsers while keeping its data-driven interaction features fully available. Along with the visualization itself several components are necessary to achieve the main goals. This concept should provide a possibility how a workflow could be to visualize a dynamic growing database holding geospatial data.

The components needed for this workflow are a main storage keeping the original data, a processing component for optimizing the geospatial features for visualization purposes and the actual web application to explore and interact with this data. The server components are all reachable using well defined network interfaces.

The next subsections are describing in detail every of these component concept ideas. Figure 5 show an overview how the components are connected.

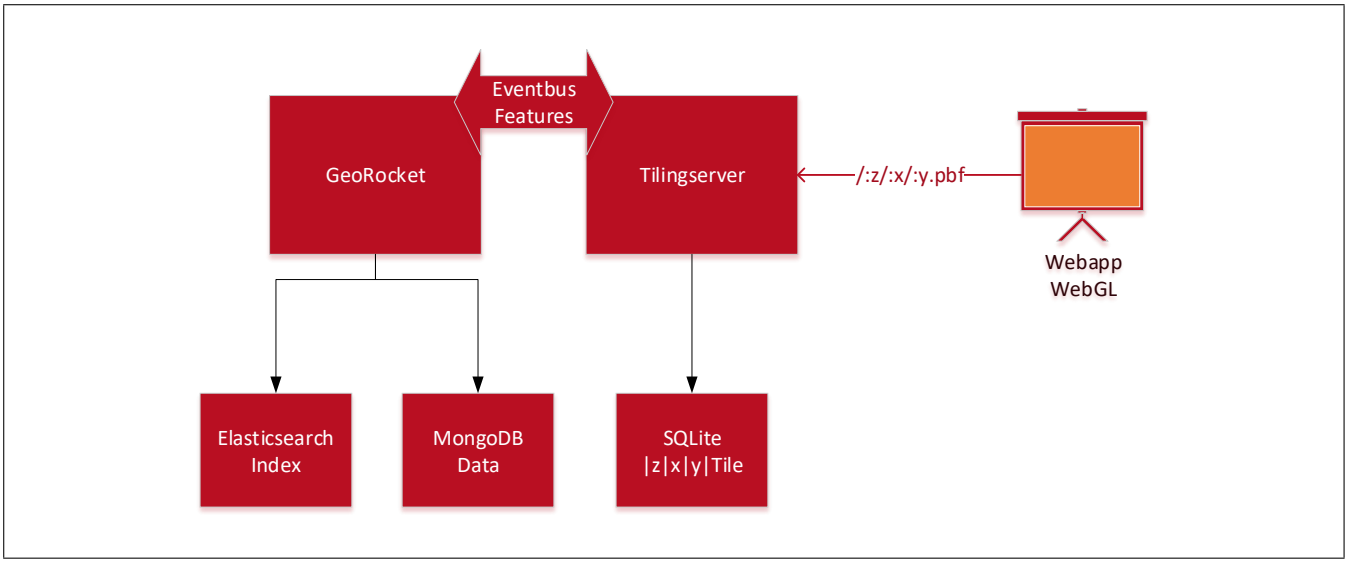


Figure 5: The three conceptual components

5.1 Main Storage

The base is a main storage for the whole geospatial data. A customer hands over files containing geospatial features in a certain format which should be integrated into the environment. The main storage consists of two components. The first one is a database storing the original geospatial features. Secondly these features are indexed to have easy and efficient access to this base system.

The customers files have to be converted into GeoJSON first to have a homogenous starting point. This means all geometry coordinate are transformed to the WGS 84 standard which is the OGC defined standard for greenwich projected longitude and latitude values. Every data attached to these geometries is added as key value property store and bundled to a GeoJSON feature [10]. These are put together to a feature collection.

To keep data integrity this approach assumes features have either unique identifiers to reference or the storage solution is maintaining identifiers for every feature. As the conversion of data is only changing the file format the original data is reconstructible at any time even without these unique identifiers. Coordinates are transformed but kept in the same precision. Though nearly no information from the original coordinate reference system values gets lost.

These GeoJSON files are imported into the main storage. Every feature is stored as a binary. The geometries are indexed by using a spatial indexing data structure like a QuadTree. Every property is indexed as well for better query and aggregation access.

This storage has to be able to store very huge amount of data to reach the scalability requirement.

5.2 Optimization Processing

To visualize data from the main storage in modern web browsers it is necessary to optimize it and store it using a efficient and specialized format. This format should be widely used and rendered by well known map application frameworks in its basic functions.

In that approach the concept of a secondary datastore will be implemented. This means it maintains references to the original feature in the main storage whenever transforming or mutating geometries. Whenever data is imported in the main storage every secondary datastore will be informed which features are newly imported and are obviously missing in all secondary datastores.

The next paragraph will describe how the optimization conceptual works.

The above described algorithm to create a tiling tree is working better having huge batches of geometries. The reason for this is, every time the tiling process is started with a new set of geometries the whole tree has to be traversed, and clipping has to be done way more often. Furthermore updating a tile is very expensive.
To optimize the number of tile tree creating batch processes about 20,000 features are buffered before starting a new process. Then the recursive tiling process is starting as described in 2.3. When a feature collection is ready to be stored as a Vector Tile one can directly store it when no old tile is existing in its particular x, y, z coordinate. Otherwise the old tile has to be decompressed, decoded, merged with the new one, encoded again, compressed and stored into the database.

These tiles encoded as Google Protobuf and then compressed using GZip need less storage space and of course in-memory space. But more important they are fast to transfer over the internet.

This is the idea of having a dynamic growing vector tile storage while maintaining this as a secondary datastore kept in sync with the main storage.

5.3 Visualization Storage

Vector tiles created by the processing approach in the last section have to be stored. They are encoded in a binary format along with three coordinates identifying every tile. All approaches are storing these tiles with the regarding x, y and z values. To render the geometries only this information is needed. The resolution of a particular tile is stored in itself.

Having tiles in place, these are served as a Tiled Map Service (TMS). The TMS interface is defined well by the OGC and very simple. The only parameters needed are the three coordinates x, y, z and the file format extension which should be served. In this approach this can be Mapbox Vector Tiles (.mvt), Google Protobuf (.pbf) or simple geojson tiles (.json).

This work will not discuss technical solutions in this section. More detailed information how to store tiles can be found in the implementation section.

5.4 Visualization

Lastly the vector tiles are visualized in a canvas using modern web browsers. Render performance is achieved because the geometry coordinates are transformed already. The integer only coordinates are living in their virtual extend [2]. The renderer only needs to transform them by scaling and adding the correct offset. Now lines can be drawn directly on the screen.

Several javascript frameworks exist to build map applications on top. They all have in common their main architecture which is working with data sources and then attaching these to layers. The main layer for our purpose is a layer displaying pre-rendered street maps or tiled satellite images.

The frameworks are providing a user interface to explore the world using a computers mouse. While scrolling and zooming through the world it requests tiles for the particular level of detail and matching the current viewport of the world. These are rendered at the correct place on top of the base layer mentioned before.

For performance optimization tiles should be dynamically cached. This depends on the size of the tiles rather than the number of tiles. A tile is growing in size very fast on adding more properties to features because the key-value store is kept in memory for every single feature. The framework has to make sure the process used by a browser is not allocating too much memory. Processes using over two gigabyte of memory are mostly killed and the web application is reinitialized.

Additionally two interaction features should be implemented.

The first one is filtering feature by properties. These property values can be strings, numbers or timestamps. This feature is very important as users not often want to have the whole dataset on screen and they often work with temporal based data in where mostly on snapshot is relevant at once. Performance can increase on adding more filter conditions. The renderer then could encode and compress many features not rendered onto the screen which saves a lot of working memory.

Another feature to implement is data-driven colorization of the geometry filling. To achieve this at least the one numeric property used for color interpolation has to be attached to the feature directly. This is necessary because the renderer needs direct access to the correct color while being in the graphics acceleration units context.

6 Implementation

In order to evaluate and proof the concept described so far the next step is to implement the proposed workflow. As mentioned above this approach consists of three main components. These are the main storage, the specialized storage for visualization purposes along with the optimization process and the visualization along with a server component.

As existing technologies are written in Java everything implemented along this research has to be integrated into an environment. This leads to the decision to do implementations in a JVM language. All written code is ether in Java, Scala or Kotlin. Kotlin is a relative new language founded by Jetbrains which glues benefits used in Java and Scala together without re-inventing the wheel or trying to build a 'eierlegende Wollmilchsau'[25]. Kotlin supports object oriented and functional programming. They have also a extension for the reactive programming pattern currently popular when working with huge data streams.

Every component itself is independent and can be used standalone. In purpose to have a working system they can easily work together. This is described in detail later in the component integration section 6.5.

6.1 Main Storage

The idea described in the concept section is mainly implemented in GeoRocket. At the starting point of this research GeoRocket was stable and useable to store and index a massive amount of data. During the working progress GeoRocket was then used as main storage along with a MongoDB as storage solution.

Some adaptions were necessary in order to access the data as needed. Whenever a specialized adaption is done, this is mentioned in the next subsection at the particular paragraphs. Now assume a running GeoRocket instance filled with geospatial data in GeoJSON format along with a lot of properties. At least on property of every type, which are numbers, timestamps and simply strings.

6.2 Optimization Processing

At first features stored by GeoRocket have to be passed to the processing component maintaining the visualization storage. To do so GeoRocket is extended by a listener, buffering features during the import process. As mentioned in the concept section at least 20,000 features are buffered. These are transmitted in GeoJSON format over a REST interface. The tiling server component then reads features from this stream and converts them into a kotlin data type in order to perform the next computation steps. Coordinates are transformed to fit into the range of zero to one.

The configuration includes a minimal and maximal zoom level. Tiles are created and stored only in this range. Even when not storing zoom level zero, the algorithm starts at this level of detail.

The perquisite wrapping the features overlapping the 180th meridian is nothing but three clipping processes. This process is also shown in figure 6. The first clips the geometries at -1 and 0 in x direction, the second at 1 and 2 also in x direction. Lastly everything is clipped at 0 and 1. Figure 6 illustrates these three feature collections. The first is the left world, the second the right world. Now the left world is shifted by 1 and the right world shifted by -1. Then both are concatenated onto the center world clipped at 0 and 1.

While clipping geometries at a certain value a buffer has to be added or subtracted in the correction direction. Figure 7 and 8 illustrates how this is done. A geometry overlapping a tile boundary results in two polygons. A straight line is created in both polygons exactly at the tile boundaries. Adding a buffer in the correct direction lets disappear these straight lines while rendering the tile only in its given extend. Users then see a complete geometry as originally imported.

Now the actual tile creation process starts. Tile $0/0/0$ can be created directly. In beforehand the coordinates have to be converted into the extend of one tile which is 4096 by default. Assuming a tile with coordinates x, y and zoom level z every point is transformed using equation 1.

$$(\lfloor extend * (xVal * z - x) \rfloor, \lfloor extend * (yVal * z - y) \rfloor) \tag{1}$$

Now all coordinates are integer values only.

To have less complexity while rendering the geometries the Dougles Peucker algorithm for line simplification can be applied. It a lightweight process to remove spikes and small corners using a precision parameter. On lower zoom levels

[25] Eierlegende Wollmilchsau - http://blog.inkyfool.com/2013/11/eierlegende-wollmilchsau-perferct-animal.html

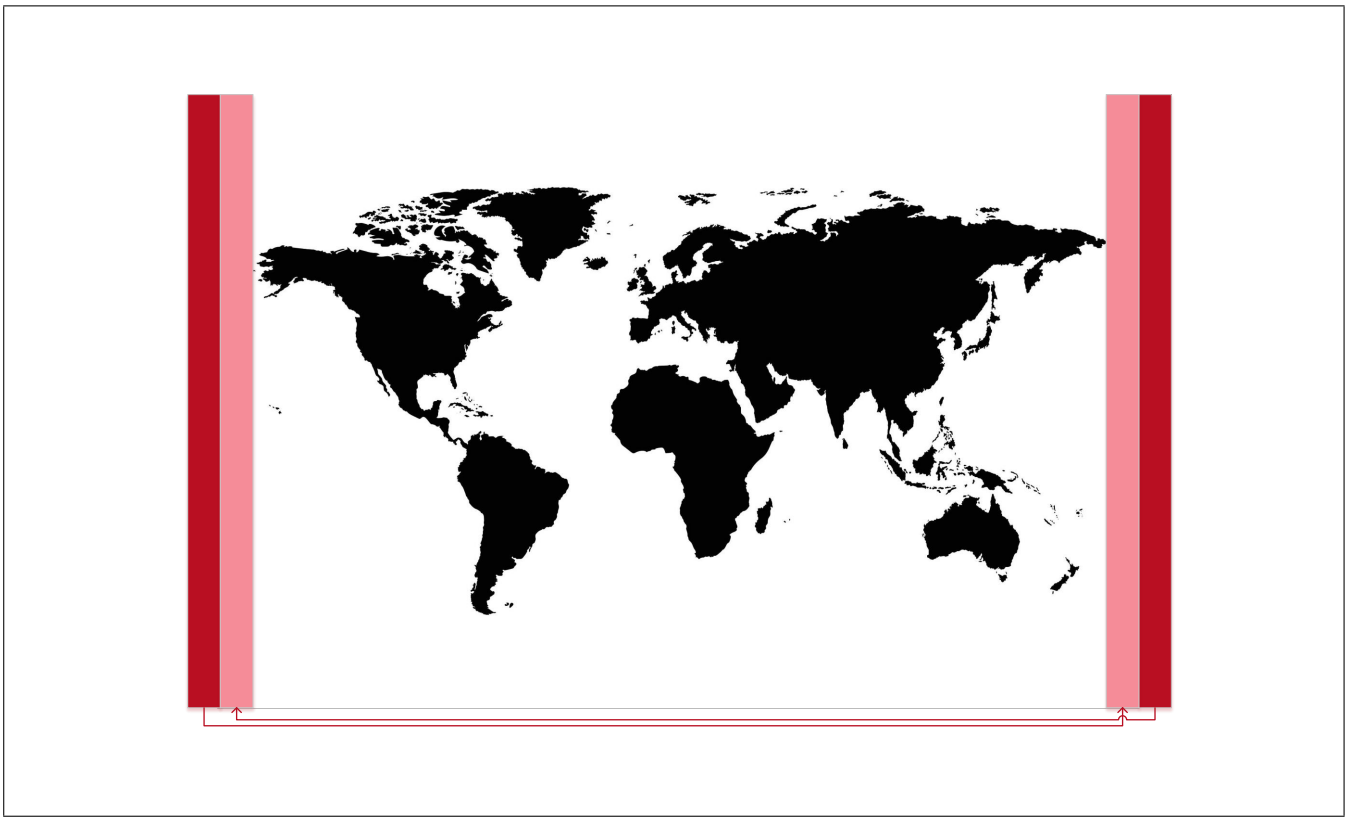


Figure 6: Wrapping features overlapping the 180 degrees meridian

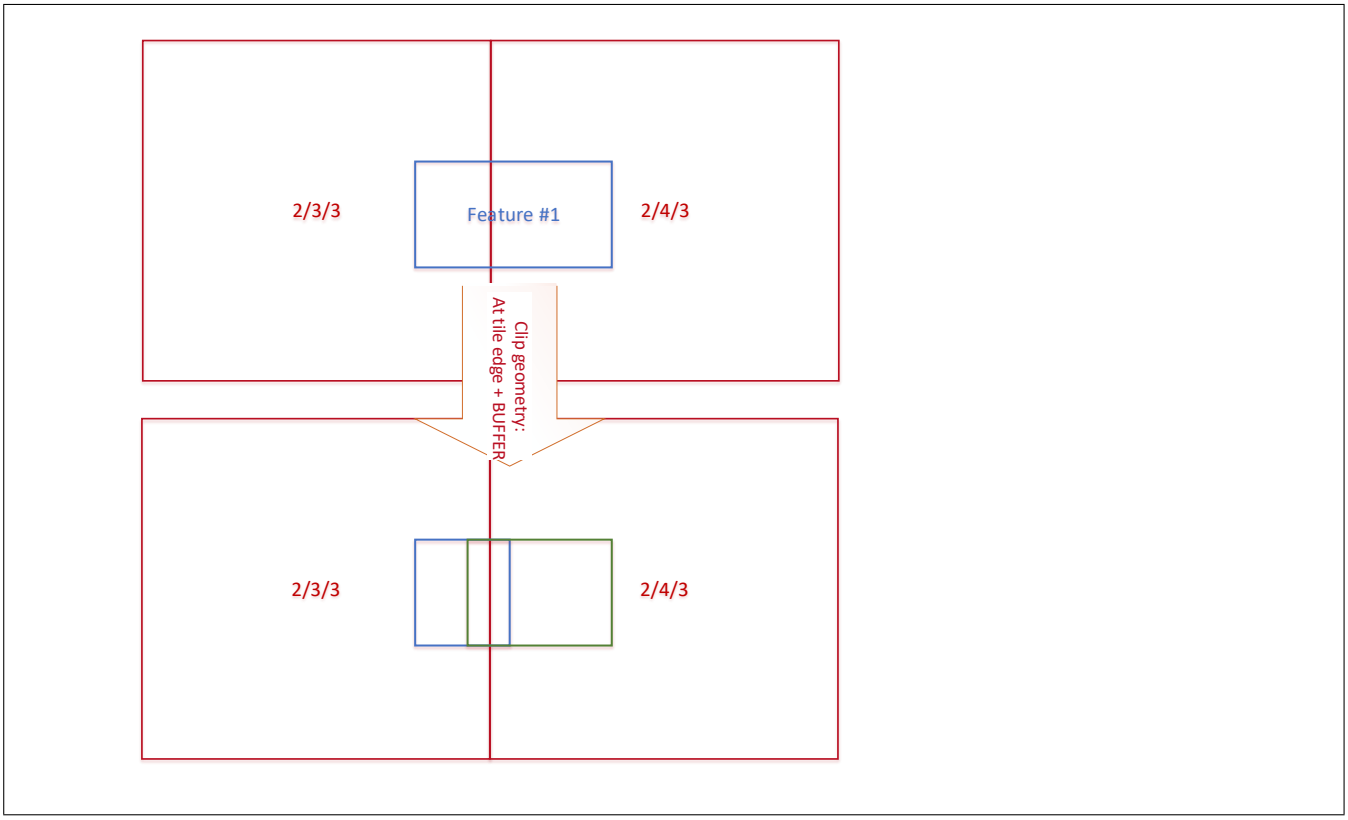


Figure 7: Split feature at tile edge and clip while adding a buffer.

these are too small to be considered by a user.

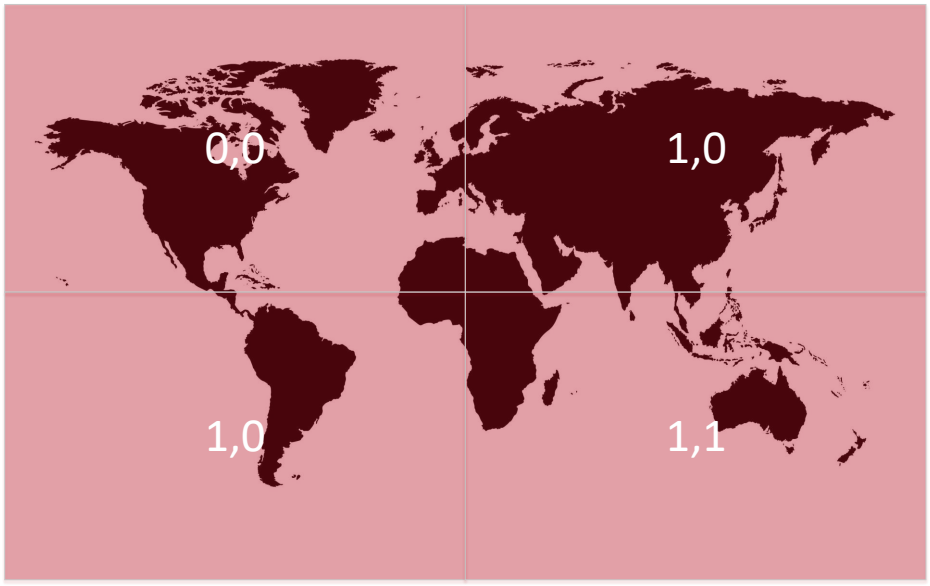


Figure 8: Tiling in zoom level 0

To encode features as Mapbox Vector Tiles polygon are represented in a efficient way. Figure 10 illustrates how this works. This simple process takes a starting point and then the vector to the next point included in this polygon until the last point equals the starting point. Assume a starting point $[x0, y0]$ and the next point $[x1, y1]$ the vector stored is calculated as stated in equation 2.

$$[x1 - x0, y1 - y0] \tag{2}$$

The geometry properties are converted to a tag set and stored into the tile as this is more efficient and does deduplication. Google Protobuf then does its work to serialize the tile which is ready to be stored.

Every tile stored is split into two equal new ones and the geometries are clipped at their boundaries. This recursive process is done until the maximum zoom level configured is reached.

During this process simple optimizations are done. Before iterating over all features to clip them, features are ignored having a bounding box outside the actual tile. Furthermore, a process is stopped completely for the current tile when the feature collections bounding box is outside of the tiles extend.

Listing 1 shows the recursive algorithm of tiling.

6.3 Vector Tile Storage

The second sub-component of the tiling server is the storage implementation itself. The internal interface is kept very generic and provides methods to update, insert and delete tiles. A store client is implemented for several backend technologies. It is configurable using a configuration file. Supported is a SQLite, Mongo and H2 Database at the moment.

The insert and delete operation is straight forward to implement. On inserting the fields x, y, z is used a the primary keys for SQLite or transformed to a unique hash (equation 3) when a key-value store like H2 or MongoDB is used. The coordinates can also be used to delete a selected tile completely.

$$(((1 << z) * y + x) * 32) + z \tag{3}$$

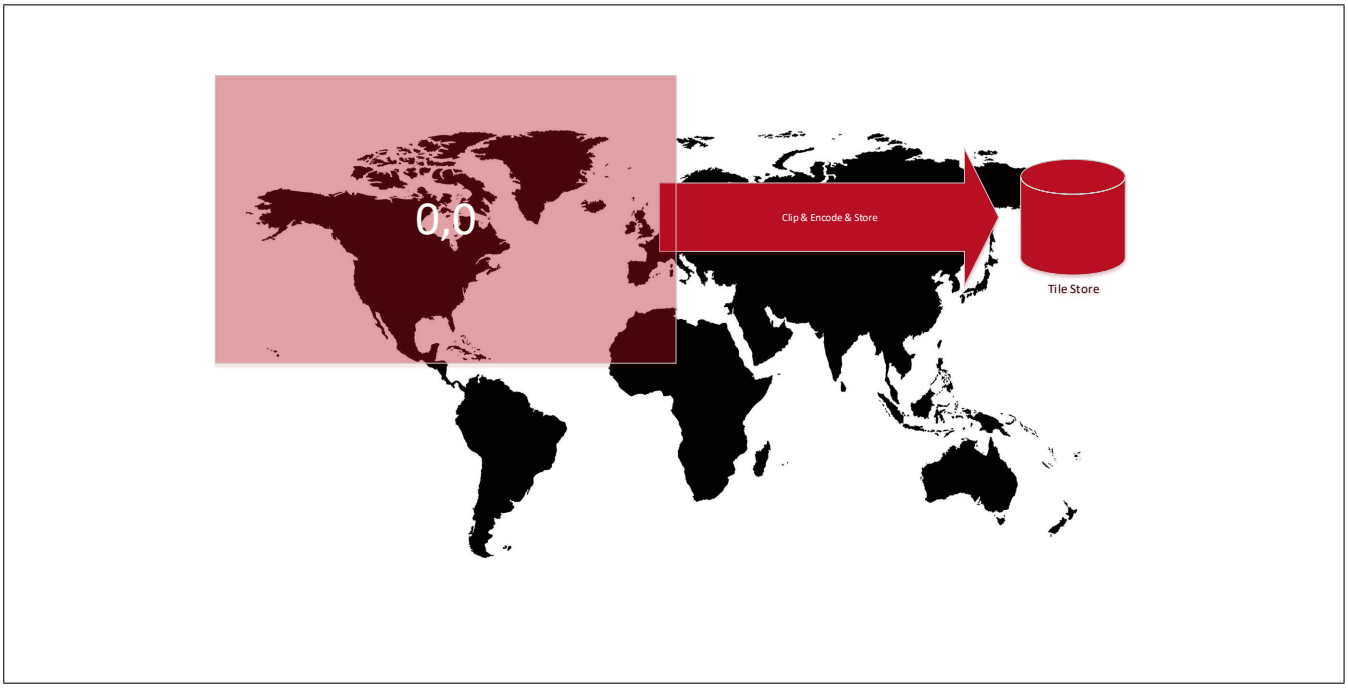


Figure 9: Store all features into tile 1,0,0 (z, x, y)

The process of updating an existing tile is not complex itself but consumes additional computation time. The old tile has to be fetched from the database and mapped into the tiling servers data type. New and old geometries have to be merged in one new geometry collection. The new properties have to be integrated into the old tag-set. Then the new feature collection must be stored.

In case the MongoDB storage is used, a garbage collection process must be implemented as MongoDB at its GridFS extension implements versioning instead of directly replacing binary data fields.

The store client then can also be used to serve tiles. The integrates tile map server (TMS) provides a RESTful interface. The parameters are x, y, z and the chosen format. Equation 4 show the REST scheme used. File formats supported are $.mvt, .pbf, .json$.

$$/:z/:x:/:y.:fileformat \tag{4}$$

6.4 Visualization

The actual rendering is done by a map application framework. There are two well maintained and feature rich frameworks supporting Mapbox Vector Tiles directly in a stable version. The most popular one is OpenLayers. Rendering Vector Tiles is implemented quite well. Users can select features and customized styling is available, too.

Main features defined in the research goals are missing. Filter features is not supported directly and data-driven styling is not supported directly, too. OpenLayers has no support for plugin extension in its newest major version (4.0). Additional features though have to be implemented in OpenLayers directly. Due its lack of performance caused by a wrong caching implementation OpenLayer was not the choice to evaluate rendering of millions of polygons.

In this approach the implementation from MapBox is used. They support only WebGL enabled browsers and provide all features needed. Caching is working as well. The tiling servers REST API is configured and a openstreetmap base layer is added. Now feature are simply rendered onto the screen.

In our implementation, developers can edit a configuration file to set filters matching the property values of the features fetched from the backend. Also two colors and a minimum and maximum value matching a property for data-driven styling can be applied. Colors then are interpolated and mapped onto the geometries. Users then can explore the world including a data-driven styling (see figures 14, 15).

Figure 10: Encoding a geoJSON feature collection in mapbox vector tiles. Source: Mapbox - `https://www.mapbox.com/vector-tiles/specification/#encoding-attr`

Speaking only about rendering a huge amount of geometries in this implemented environment works quite well. Limitations and issues as well as further ideas for research and implementation are discussed in the evaluation and future work section.

6.5 Component Integration

The last sections have described the components needed itself. Every server component can be used standalone.

The main storage can be used to efficiently store and index geospatial data. The tiling server component can import GeoJSON files on its own. Also one can use the tile storage to maintain a tile tree by himself and just use the map application frontend to visualize the tile tree and explore the data.

In this approach it is important to see all components together. The data flow should go through all components and even back to the originally imported data.

The starting point are files containing geospatial geometries and attached data. Assuming the files first have to be converted to a suitable format. For instance a CSV file is then converted by a plugin plugged into the main storage. After that the main storage is indexing and storing the data split up into features. Every feature imported releases an event one

Algorithm 1 Recursive tiling

```
MIN_ZOOM ← 0
MAX_ZOOM ← 17
EXTENT ← 4096
BUFFER ← 64
Q ← emptyQueue

function tile(SetOfFeatures)
    wrappedFeatureSet ← Wrap(SetOfFeatures)
    q.offer(wrappedFeatureSet, 0, 0, 0)
    while q.isNotEmpty do
        t ← q.poll
        split(t.features, t.z, t.x, t.y)
    end while
end function

function split(SetOfFeatures, z, x, y)
    zs ← 1 << z
    if z >= MIN_ZOOM then
        Tile ← createTile(SetOfFeatures, z, x, y)
        updateTile(z, x, y, Tile)
    end if
    k1 ← 0.5 * BUFFER/EXTENT
    k2 ← 0.5 − k1
    k3 ← 0.5 + k1
    k4 ← 1 + k1
    left ← clip(SetOfFeatures, zs, x − k1, x + k3, 0)
    right ← clip(SetOfFeatures, zs, x + k2, x + k4, 0)
    if left.size > 0 then
        top−left ← clip(SetOfFeatures, zs, y − k1, y + k3, 1)
        bottom−left ← clip(SetOfFeatures, zs, y + k2, y + k4, 1)
    end if
    if right.size > 0 then
        top−right ← clip(SetOfFeatures, zs, y − k1, y + k3, 1)
        bottom−right ← clip(SetOfFeatures, zs, y + k2, y + k4, 1)
    end if
    if z <= MAX_ZOOM then
        if top−left.size > 0 then q.offer(Tile(top-left, EXTENT, x * 2, y * 2, z + 1))
        end if
        if bottom−left.size > 0 then q.offer(Tile(bottom-left, EXTENT, x * 2, y * 2 + 1, z + 1))
        end if
        if top−right.size > 0 then q.offer(Tile(top-right, EXTENT, x * 2 + 1, y * 2, z + 1))
        end if
        if bottom−right.size > 0 then q.offer(Tile(bottom-right, EXTENT, x * 2 + 1, y * 2 + 1, z + 1))
        end if
    end if
end function
```

can read in order to do further computations. In this approach we maintain a optimized datastore one has to keep up to date. Though, these newly imported features are fetched from the main storage and the data gets through the tiling optimization process. Finally the attached tile tree storage stores or updates every tile when needed. At last the web application receives an event a tile is updated and has to be reloaded. Now the visualization is updated.

Technically there are two methods all components are communicating with each other.

The first more traditional way are REST interfaces. Mostly listeners are polling for updates at the other components REST interface. This approach is straight forward to implement, easy to maintain and debug. It produces a overhead in

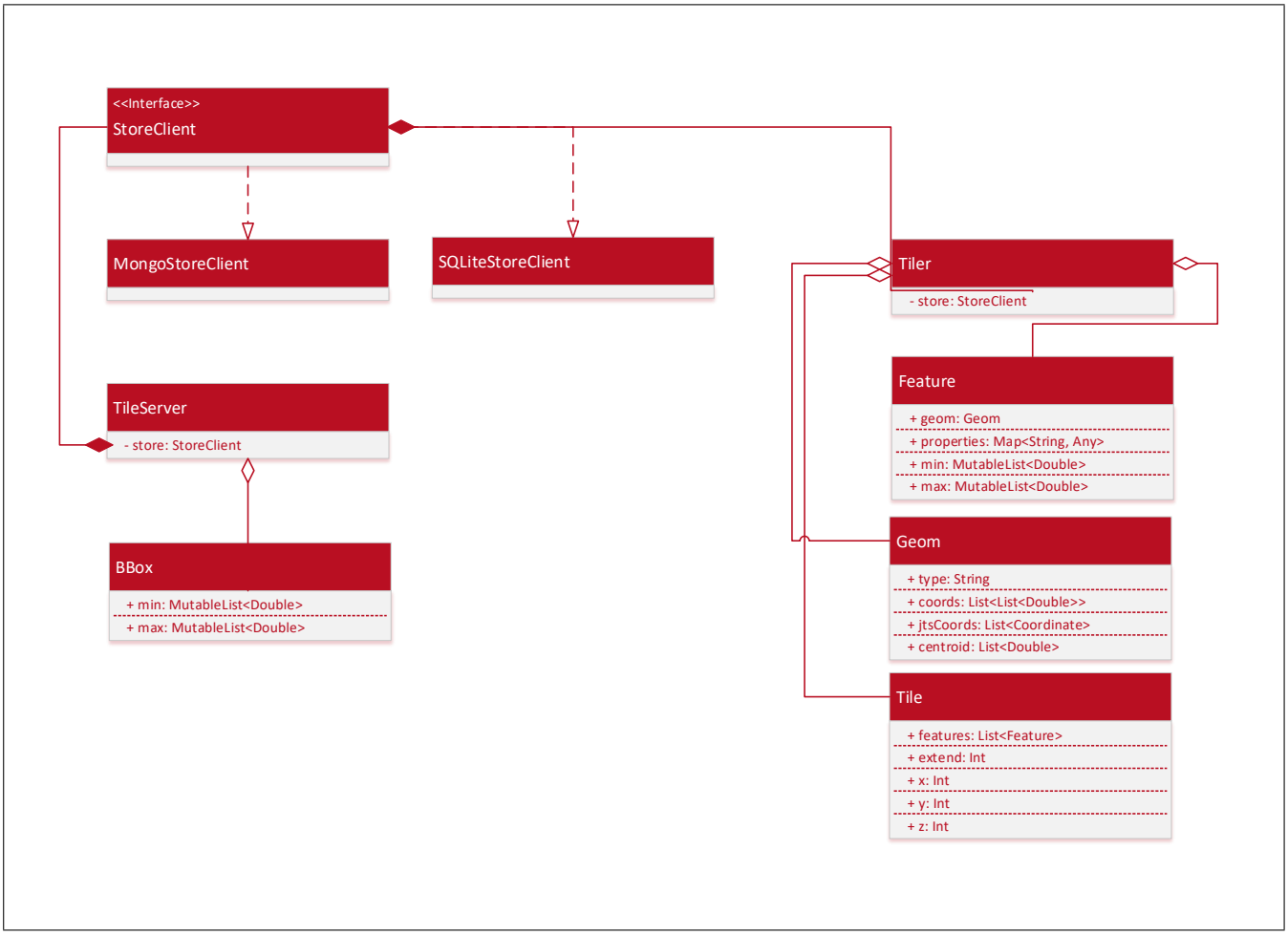


Figure 11: UML class diagram of the tiling component including storage and server.

network traffic, and one component needs some time to get the update notification.

The more advanced approach is to implement an eventbus. In this implementation the vert.x reactive framework is used. They implement an eventbus using the in-memory grid filesystem HazelCast. Using websockets and multicast protocols a multi-directional communication is possible. This implementation allows to get fast information about importing, indexing and tiling updates.

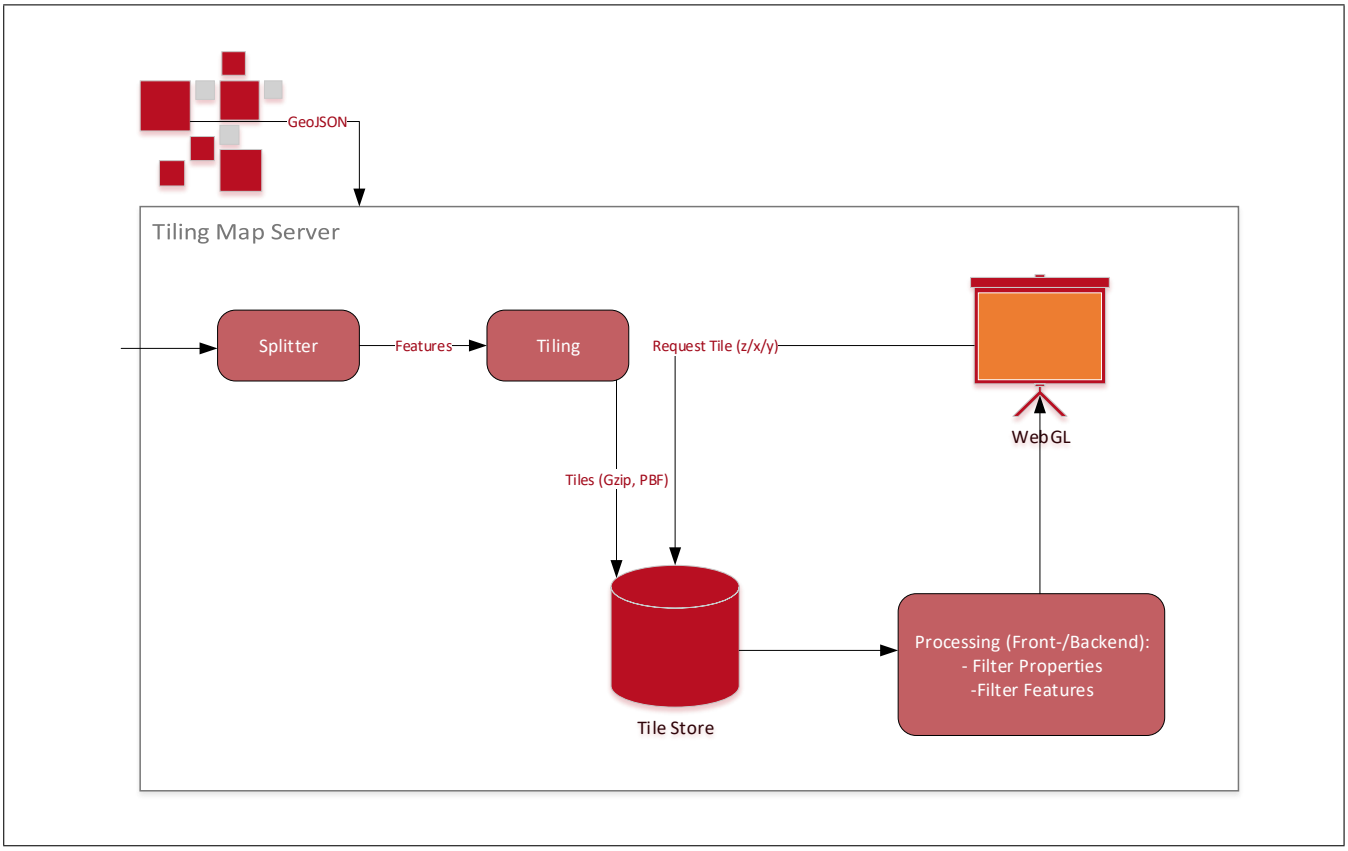


Figure 12: Tile Map Server Concept (TMS)

7 Evaluation

This section should point out how well this first approach works conceptual as well as the implementation. More important, issues and limitations appeared during the research are mentioned and how they could be solved. These ideas and solutions are described slightly more detailed in the future work section 9.

The evaluation also references before stated goals and requirements.

7.1 Geospatial Storage & Index Solution

The starting point of this research was focussed on the visualization aspect directly. GeoRocket was used to store and index data for further computations and optimization processing purposes. Is this particular task, GeoRocket is very fast and efficient, but there are limitations not recognized in beforehand.

Elasticsearch is used as indexing framework. The focus of indexing in Elasticsearch is not the geospatial aspect. Elasticsearch is more a software to collect a massive amount of sensor based data or to index a huge catalog of communication messages in order to do data analytic task on those. Usage as a search engine is another huge use case Elasticsearch performs well. Additionally geospatial information can be attached.

Missing are more complex data structures and configurable parameters for spatial indexing. Furthermore, spatial operation and aggregation possibilities are not available at all. Ideas in another directions more focussing on the visualization and the geospatial aspect are stated in the Future Work section.

Important to mention here is, GeoRocket including ElasticSearch can be used very well as a really basic, reliable and fast data storage. To do more then storing data it need a more specialized solution.

The important requirements performance and data integrity are achieved regarding the main storage. Performance tests are done already here by Krämer [8]. Focus was visualizing and not particular storing data.

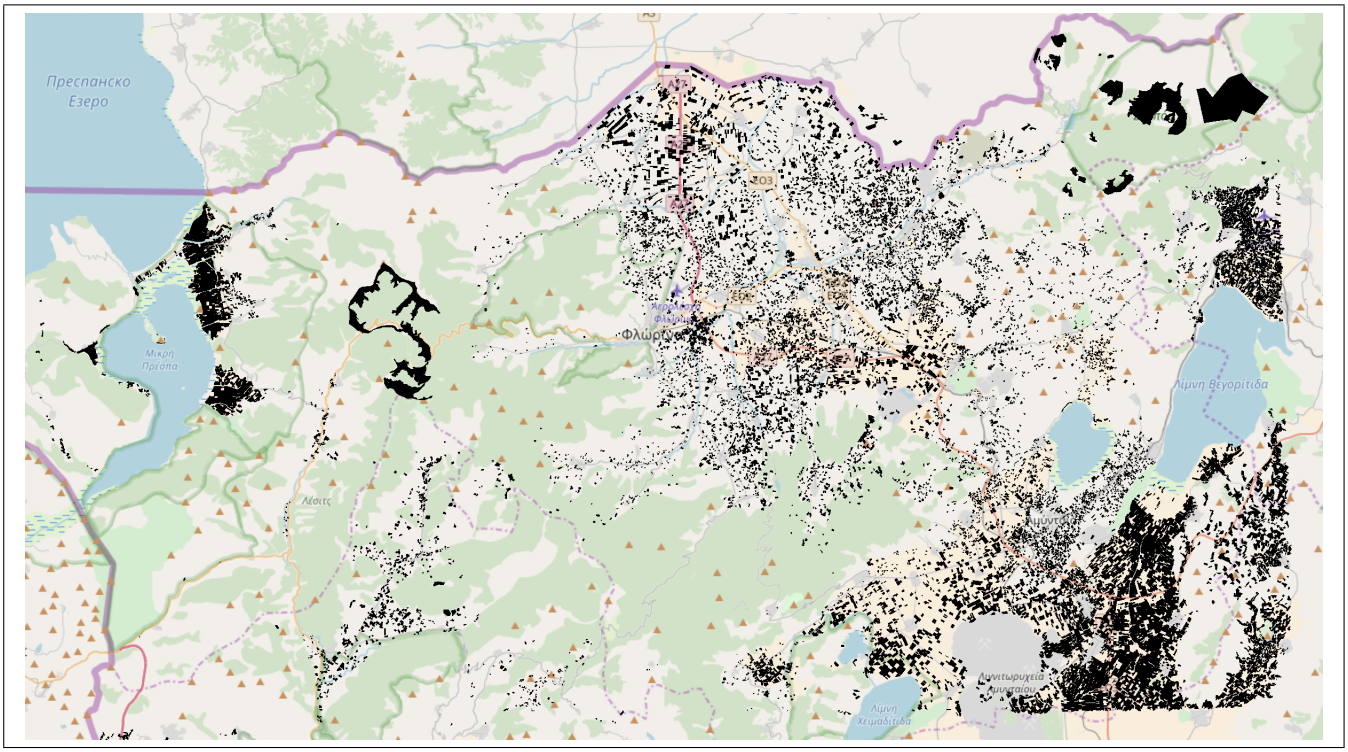


Figure 13: 500,000 Parcels loaded into memory and rendered using MapBox GL JS

7.2 Tiling Server Component

Included in the tiling server component is the process optimizing data for visualization purposes and the Mapbox Vector Tile storage serving the tiles as a RESTful service.

7.2.1 Processing Time

Processing time is important because data is not updated twice a year, but once a day probably. Users are importing data and want to explore data using the map application as fast as possible after importing new datasets. Though the important time to measure is from the starting point of the tiling algorithm until all features are stored in tiles into the highest level of detail. Two ranges of zoom levels were configured during the tests. At first zoom level 0 to 10 was tested, and afterwards the full range to zoom level 17, which is the highest standard map applications are presenting to users mostly.

The performance was tested using real and synthetic datasets already converted into GeoJSON format including WGS 84 coordinates.

The real dataset consists of about 500,000 agriculture parcels containing the parcel itself as a polygon and data describing the type of the parcel and several quality indicators.

The synthetic datasets contain a specified number of polygon with a given radiant and a given number of vertices. One approach is to position the polygons randomly over a given bounding box onto the world. The other way is to homogenous spread the data over a the bounding box.

The measurements are showing the time needed after reading one data portion in particular one file into memory until storing all tiles to a given level of detail is finished. Time needed for reading files into memory is skipped here because it is not representing the speed of building the tile tree. It only states the speed of the used hardware and the JSON parsing implementation.

Table 1 shows processing times for the synthetic datasets. Table 2 is showing the process of importing the one real file provided in context of the DataBio project.

These results are revealing a main problem in this implementation. Creating tiles from zoom level 0 to 10 can be done in a few minutes even for over a million polygons. Also they can get complex and still created in a reasonable time users

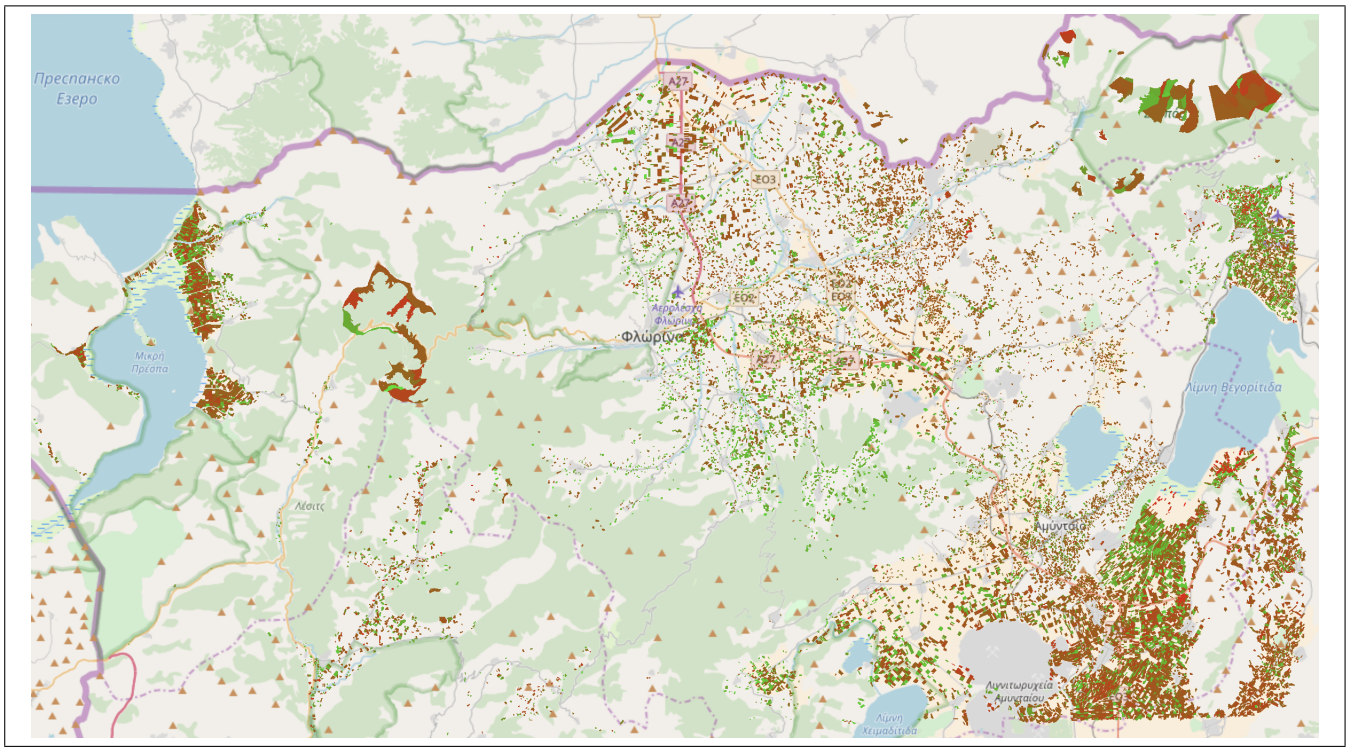


Figure 14: Same parcels as in figure 13 but with data-driven styling

# of polygons	Vertices per Polygon	highest LOD	Processing (in min)
1,000,000	4	10	1.80
500,000	10	10	2.17
100,000	50	10	1111
100,000	50	17	14,31

Table 1: Processing Results Sythetic Data

could effort to wait.

Creating tiles in more detailed zoom levels takes non linear more time. There are several reasons why tile creation is getting very slow especially on zoom levels over 13.

One limiting factor is the used storage solution and the hard disks used to store tiles into. As explained before MapBox uses SQLite as a their default solution for storing the vector tiles itself. The benefits of SQLite are obvious for static datasets. Everything stored in one file, clients are very easy to implement and the data structure is not complex at all. But a transaction in SQLite is blocking till its finished. On higher zoom level their could be millions of tiles which have to be inserted into the database. This can be regognized very easy when displaying how much tiles are inserted per second while processing.

The concept of updating existing tiles was explained in detail before. This leads to even slower transactions. Using one of the other storage solutions makes it faster but not as much to do further performance tests using MongoDB or H2.

The question to answer in further researches is, does another solution to store tiles directly whenever new data appears exist or could a on-the-fly tile creating solution be better in combination with another main storage technology as stated in the subsection before.

7.2.2 Disk Space

For now the fact the secondary datastore need space for data already available in the main storage because optimization is more important then wasting disk space. At first the actual size of the tile storage is measured using gzip compressed vector tiles.

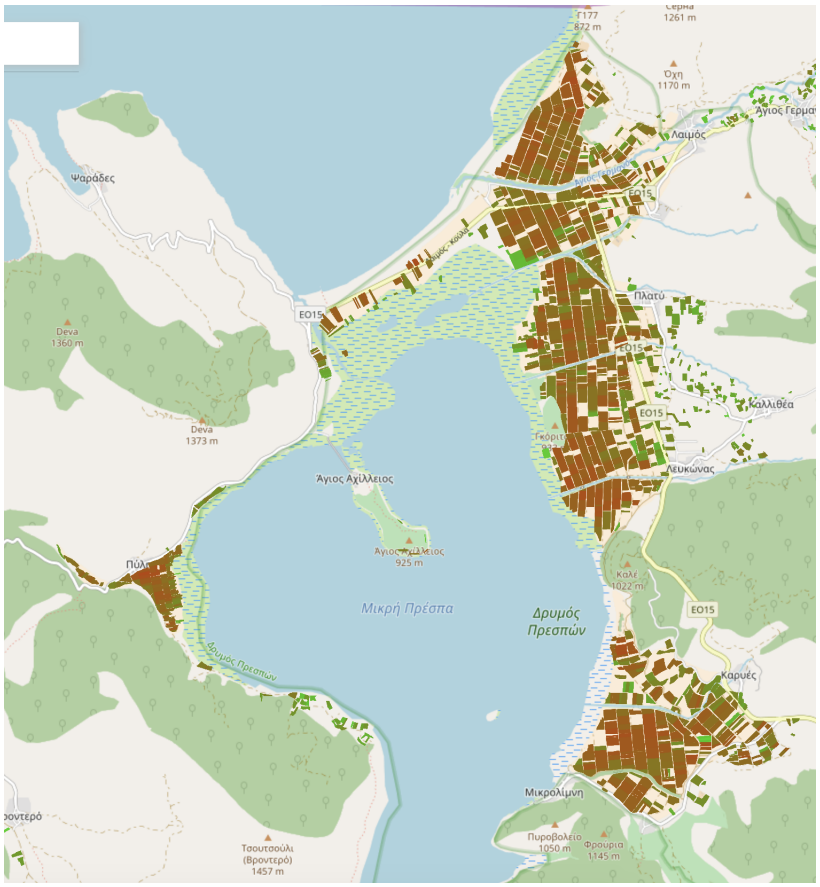


Figure 15: Higher level of detail data driven styling

Figure 16: User interface to apply data-driven colorization

The synthetic GeoJSON files have a size of 200 to 500 megabytes.

Database

Firstly disk space used by the SQLite database file is discussed.

After the tiling process the SQLite database uses about 200 to 250 megabytes of disk space. The main factor is not the number of features included in the imported files. Mainly the number and size of property values need more disk space.

Consider a property key-value store as stated in figure 10. Every key is stored only once per tile. Every value is stored with its tag identifier. Every value is only stored once even when its attached to different features. Furthermore, tiles are compressed using gzip at the end.

Using this knowledge one can consider more disk space is only used when adding properties having different values for each feature. In most cases this is only the case when having complex information like a binary array as property value. These should be avoided for several reasons. Obviously more disk space is allocated, but more important it blows up memory when decompressing and decoding tiles. This is explained in more detail in section 7.2.3.

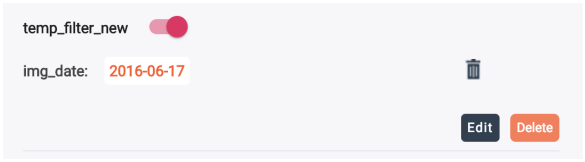


Figure 17: User interface to apply data-driven filters

# of polygons	highest LOD	Processing (in min)
500,000	10	2.17
100,000	10	1111
100,000	17	14,31

Table 2: Processing Results Real Data

Duplication

The next important topic when evaluating disk space is duplication of data portions.

So far no spatial aggregation or more complex spatial simplification method is introduced. So, every feature is saved at least once per level of detail. When a geometry overlaps tile boundaries all properties are saved once per level of detail and additionally once per tile the geometry is included.

Duplication grows when importing more geometries covering a huge part of the world. A geometry covering for instance germany completely is easy to store in zoom level 0 to 10. Going further every tile covering germany has to be stored including the properties attached to the one geometry. These are about 100,000 tiles on zoom level 17.

Duplication can not be measured using vector tiles using protobuf and gzip. This eliminates duplication per tile completely. For this purpose tiles are stored as GeoJSON feature collection in the same way as mapbox vector tiles but just plain text into the SQLite database.

The real dataset includes 500,000 features having a huge key-value store attached. One field is even a binary array with a length of 10. Splitting this GeoJSON into tiles, one can clearly see duplication is done exactly as stated before. This file only contains very small geometries, so every geometry is stored once per zoom level mostly. Measuring the disk space used per zoom level every zoom level consumes the same amount of space. The disk space was computed using the SQL statement in listing 1.

Listing 1: Disk space computation using SQL

```
1  SELECT zoom_level, (SUM(length(tile_data))) as size
2  FROM tiles
3  GROUP BY zoom_level;
4
5  SELECT MAX(zoom_level) as max_lvl, (SUM(length(tile_data)) / 1024 / 1024) as size
6  FROM tiles;
```

Now one can assume a geometry covering germany duplicates data exactly the same way.

The conclusion in case of disk space usage is, duplication happens in the manner of the tile creating algorithm. But the encoding scheme eliminates duplication at least per tile. Though, in manner of disk space it is not a big issue.

7.2.3 Memory Usage

This section briefly describes the memory usage during the tiling process.

Because the algorithm uses a top to bottom method all features have to be in-memory before building the tile tree. It is implemented using java which need more memory everything is an object and holds additional information per feature like bounding boxes to optimize the tiling process. After creation of one tile features included in the next level of detail are kept in an array and passed to the next recursion call. Memory is freed at the very end of the tiling process.

Tiling the dataset containing 500,000 features need about 10GB of memory. This process is not optimized at all, which can be done easily copying features for the next zoom level and releasing memory used for the last stored tile. Implementation optimization was not the focus of this work, so this is not evaluated in detail here.

7.3 Visualization

Visualization is the main purpose of this research at its starting point.

To do performance tests a simple plain javascript application using Mapbox GL JS was implemented. Interaction is one of the main requirements and is implemented in two ways. Developers are able to add filter conditions and a configuration for data-driven geometry colorization.

Firstly the evaluation is focused on exploring data using a standard map application interface. Zooming is done by using the scroll wheel of the computers mouse. Panning is done straight forward by holding the mouse button an moving the map in the wanted direction.

Properties attached to geometries are ignored for now. Network latency and bandwidth is ignored in this evaluation step as well.
Exploring a pre-processed database of 500,000 polygons, which are covering a very crowded small area in greece was not a problem. The user interface feels very responsive during fast scrolling around the map. Transmitting the hugest tile lasts about 2 seconds. Zooming onto more details zoom levels it lasts longer to transmit tiles as many requests to the database are done. These are mostly canceled when zooming fast into zoom level 17. These requests still need computation time on the server because cancellation of threads in a reactive web server is not very easy, but the transmission over the network itself is not performed.

Rendering and transmitting two million polygons having about 5 to 10 vertices is working fluently, too.

Now properties are added to each geometry and transmitted and rendered the same way as stated above. Adding one numeric value can be loaded into memory having the 2 million polygons. When scrolling around on multiple zoom levels browsers kill the process due too much memory allocated. Considering only 200,000 polygons having 10 numeric floating point values, browsers kill the process faster.

The next sections evaluate in which manner network usage and transmission time as well as memory usage in modern web browsers can limit a big data visualization using vector tiles.

7.3.1 Network Usage

As mentioned multiple times the scheme used to store vector tiles is very efficient and furthermore, can be compressed efficient as well. All experiments done pointed out network transmission is not a problem when having at least a 3G mobile data connection.

The conclusion here is, vector tiles are a very good data format to use when transmitting huge amounts of geospatial features through the internet. Even having huge duplication through the zoom levels make transmission times better then transmitting one huge GeoJSON file at once. It make the user interface very responsive considering no memory or render problems occur.

7.3.2 Memory Usage

After having done a lot of experiments using different datasets and different render frameworks the main issue with these solutions is the allocation of too much memory in modern browsers. It does not matter which browsers is used or which operating system the browser is running on. Even having hardware with 32GB of working memory available dose not change anything because browsers do not allow to allocate more then about 2GB on javascript heap space.

The workflow done by a renderer like MapBox GL JS does is illustrated in Figure 18.
As one can see caching is done by counting tiles only. Tiles are decompressed and decoded when transmitted. They are then parsed into a JSON Object to be processed in the javascript environment. Every feature needs about 1KB of disk space, which is really a lot considering there are 500,000 thousand features. Memory is then blown up by properties stored along with every feature, and even duplicated through the zoom levels.

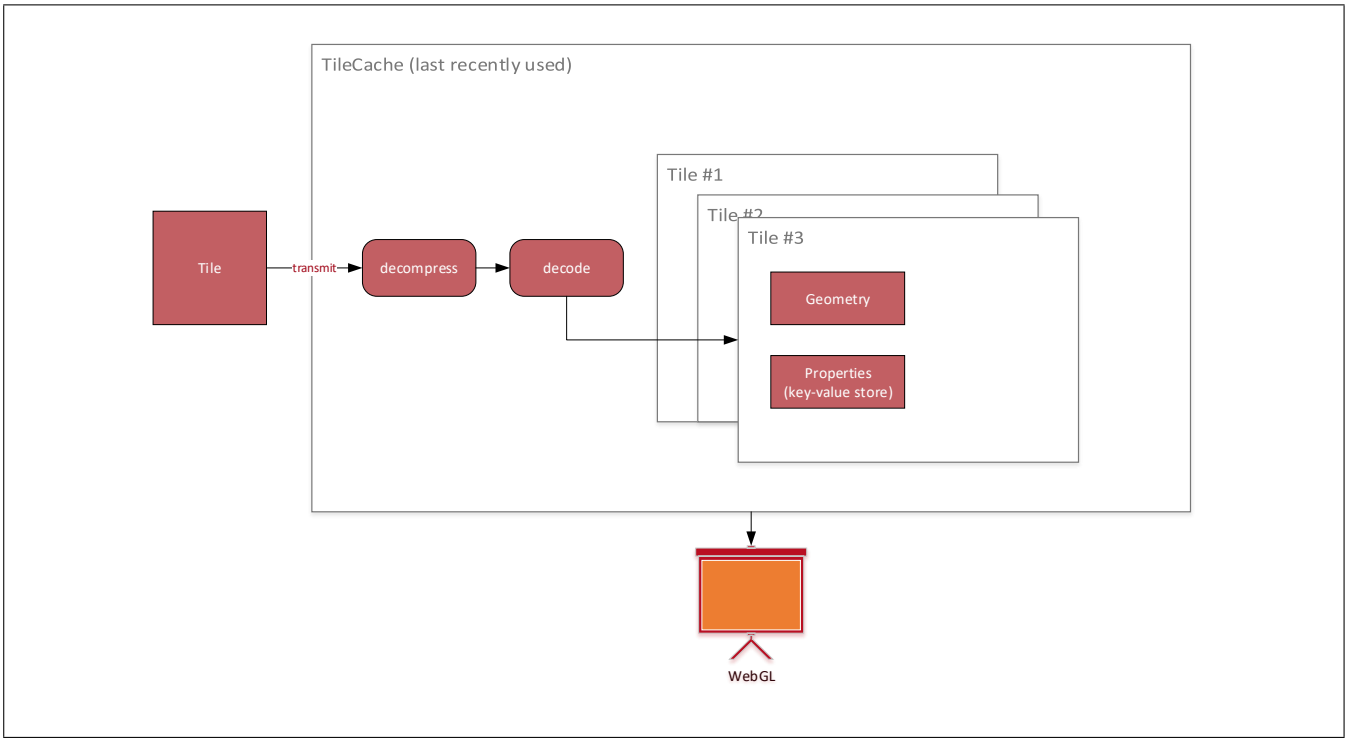


Figure 18: MapBox GL JS Cache workflow

A tile counting tile cache makes no sense here, because one tile can easily allocate up to 500MB of memory. As described before holding only geometries without properties in memory is not a problem at all. The workflow of attaching all properties straight forward directly to the features is not the right way.

Users may add filter conditions to not show all features onto the map. Currently filtering is done at frontend side, which means all features are stored still and it does not release memory, too.

During the research many ideas were developed and have to evaluated during a deeper research. Some of these are described in the future work section briefly. More experiments have to be done in order to find a suitable solution.

8 Conclusion

The focus of this thesis was to evaluate whether it is possible to provide users a platform to explore huge geospatial dataset using a interactive visualization technology. In the relevant background section existing technologies are reviewed briefly. Most important concepts used for this particular approach were described in detail. This insight was used to find a first draft solution towards the pre-defined goals. A first implementation was developed to evaluate this draft solution.

There are three main components in this approach. An essential part is the main storage holding the original data as a base to build the other components on top. For visualization purposes the data was optimized in a way it can easily be stored, transmitted over a network and rendered using modern web browsers. The last component is the map application providing a well known intuitive user interface in order to explore geospatial data rendered onto a base layer like open street maps or satellite images. Using this implementation over two million polygon were rendered. One metric was attached to the geomentries to evaluate the interactivity of the prototype. It was capabable to do data-driven colorization and filtering data on the fly, satisfying the main requirements of this thesis.

The main limitation with this workflow was memory allocation while fetching, holding and rendering spatial data using a web browser. This was caused by the properties attached to the actual spatial data (geometries), because it is not hold in an efficient way after being transmitted and cached in memory by the map frameworks. Data is completly decompressed, decoded and parsed to a JSON object. Even when currently not visible every tile is hold in memory. An adaption of the configuration to only cache about 16 tiles is not working as well, because only on tile has to be huge enough to allocate to much memory.

Further research has revealed ideas to solve this issue. As stated out in the next section about future work, optimizations should not only be done on the web application and render side. Also the currently chosen main storage is eventually not the right solution to store geospatial data in order to extend it with additional features.

The most promising and important ideas are focusing on the process of optimizing data and the used data format of storing the tiles. Also the way of serving tiles could be optimized. Users may not see all features at once, though, they must not be transmitted at all. Caching at frontend but also backend side has to be optimized as well.

This approach should be seen as a starting point for deeper research in the topics of geospatial index and storage solutions, geospatial data formats optimized for visualization purposes and technologies to provide a map user interface. All these topics have in common that state of the art solutions work very well using static data, one time pre-processed. But in todays world it is important to manage and visualize dynamically growing databases especially those storing geospatial data.

See the next section for further research and ideas in these topics.

9 Future Work

This section introduces ideas for more optimizations in order to achieve the goals stated in the research objective section.

The ideas or features are not implemented or evaluated yet. Some are briefly tested in some experiment, but nothing more.

9.1 Overview

- Secondary data store [9.2]
- On-the-fly tiles [9.3]
- Geometry simplification and aggregation [9.4]
- Multi-Threaded Execution [9.5]
- Frontend and backend cache [9.7]
- Tile manipulation while streaming [9.6]
- Dynamic Tile Resolution [9.8]

9.2 Secondary data store

As described in the concept section geospatial features imported into the main storage are then passed to the tile building server. On a growing database the tile creating process keeps the main database and the tile storage up to date.

To implement the concept of a secondary datastore completely, one has to be able to delete features in the main data storage while the secondary datastore is deleting these features, too. In this approach, that is very inefficient, but possible. Every tile possibly holding the feature or a part of the feature is looked at to delete the correct feature.

This feature is not implemented hence their was no need and deletions can not be performed productively this way. In order to have a more dynamic data structure, but also optimized for tiling, the next subsection describes the idea of building tiles on-the-fly in an efficient way.

9.3 On-the-fly tiles

One huge benefit on vector tiles is to transmit a huge amount of information using a small amount of payload without any processing time on the tile providing server. All tiles are precalculated and therefore it is easy to respond to frontend requests.

This tile tree is very a very static data structure. To be more flexible in adding and deleting data should be stored in a way, tiles can be created very efficient. Using the current solution would lead to a spatial request and then all its computations needed to create a tile.

Following steps are the first simple idea to have both benefits of the static tiling store and dynamic storage.

- Transform geometry coordinates to some virtual extend only using integers.
- Create a tree data structure like a QuadTree
- Store features in already Protobuf encoded fields
- On requesting a tile the tree is traversed and features are streamed while traversing the tree

A data structure drafted here has to be glues together with a efficient caching. Ideas on caching are described in the regarding section 9.7.

9.4 Geometry simplification and aggregation

There are many reasons to simplify geometries or even aggregate geometries together to one huge cluster.

- Less duplication
- Less data to transmit, hold and render on lower levels of detail (zoom level 0 to 10)
- Less rendering time
- Less memory needed in browsers
- Users get a better overview over their data before getting in more detail

9.4.1 Aggregation

Assume the real data provided for evaluation purposes before. 500,000 geometries were crowded together in a small part of greece. On starting the map application in zoom level 2, the only thing to see is a black chunk in greece.

Information users gather from this viewpoint is, that database hat information about greece. This is exactly the same information one would get from one polygon covering that small part of greece. But this would be more efficient to store, render and transmit.

The idea is to do automatic spatial aggregations depending certain factors. As ElasticSearch was used as indexing framework many aggregation operations exist. The issue is, they all are no spatial aggregations. One has to reconsider whether ElasticSearch is the right solution for this purpose.

Regarding on this problem aggregating the actual geometries could be useful. Three popular approaches to do such an aggregations exist. A convex hull, concave hull and a box aggregation likely used for building surroundings.

- The convex hull creates an outer line around the points
- the concave hull creates a inner outline
- boxes around a building is nearly the same as concave hull but does only allow direction changes 90 degrees

Most commonly is the convex hull because it is very efficient and easy to implement. The other two aggregation methods have a better result especially when aggregating more geometric formed features like buildings, agriculture parcels or parking lots.

As these methods are more or less straight forward, the next section gives an idea how to find the right features to aggregate together.

9.4.2 Clustering

In order to aggregate polygons subsets of features have to be found. Features therefore are clustered for every level of detail and afterwards aggregated using a algorithm mentioned above. Clustering is much easier on points only then on polygons, so some polygons are broken down to its centroid point to build clusters.

These are some method ideas to do polygon clusters. Experiments were done on all of them, but a evaluation is not part of this thesis.

- Density based clustering using centroids 9.4.2
- Density based clustering using polygon buffers 9.4.2
- Grid clustering
- Grid clustering with distance measurement afterwards
- K-Means clustering

Density based clustering

Polygons could be clustered depending on the distance to each other. One method is density based clustering for spatial datasets. Ester [4] has introduced an algorithm called DBScan.

Clustering on the polygons centroids to be more efficient, so ignore that they exist for now.

Now follows a short description what dbscan does. Assume a bunch of geospatial points to start with. Choose a random geometry to start a new cluster with. Then calculate the distance to every other point and add all points within a given threshold distance to the cluster. Add these points to the queue until and repeat until the queue is empty.

Then start over by choosing a new random point. Figure 2 illustrates the algorithm.

Algorithm 2 DB-Scan pseudo code, source: [4]

```
function DBSCAN(SetOfPoints, Eps, MinPts)                ▷ SetOfPoints is UNCLASSIFIED
    ClusterId ← nextId(noise)
    i ← 1
    for i <= SetOfPoints.size do
        Point ← SetOfPoints.get(i)
        if Point.ClId == UNCLASSIFIED then
            if ExpandCluster(SetOfPoints, Point, ClusterId, Eps, MinPts) then
                ClusterId ← nextId(ClusterId)
            end if
        end if
        i ← i + 1
    end for
end function
```

The result are good in quality but the algorithm is not very efficient especially on huge sets of point, so it has to be pre-calculated.

Grid

The other completely different approach slices requested boundary into cells. All geometries inside these cells are our clusters for now and could be aggregated to one ploygon. This leads to a similar result as the non-recursive way of clustering but is much faster.

Take these grid clusters centroids and move them to the centroid of all geometries inside that cluster. Use the count of geometries per cluster to weight these clusters and calculate a distance threshold depending on our level of detail. Then apply the recursive buffer intersection method onto these cluster centroids.

At last aggregate all geometries using these resulting clusters.

9.5 Multi-Threaded Execution

To improve performance software often is adapted to to computations in parallel. The tile tree building algorithm fits good into a multi-threading environment. Every single tile creation process runs independently. Therefore it can be executed without syncing the threads to wait for an result. Assuming the tile store transaction is non-blocking this could increase the performance a lot.

At the moment only functions operating of a list of features or geometries are processes in parallel. Doing this on the tile queue itself would be a easy and good optimization.

9.6 Tile manipulation while streaming

Figure 19 illustrates the idea of filtering whole features or single properties using user-defined conditions. As the vector tiles are encoded using Google Protobuf, which serializes the tiles, it could be possible to do so.

Further research has to be done in order to evaluate this idea. They main question is here: Is the mapbox vector tile scheme or even Protobuf itself suitable for this kind of streaming? Or needs the data format modifications?

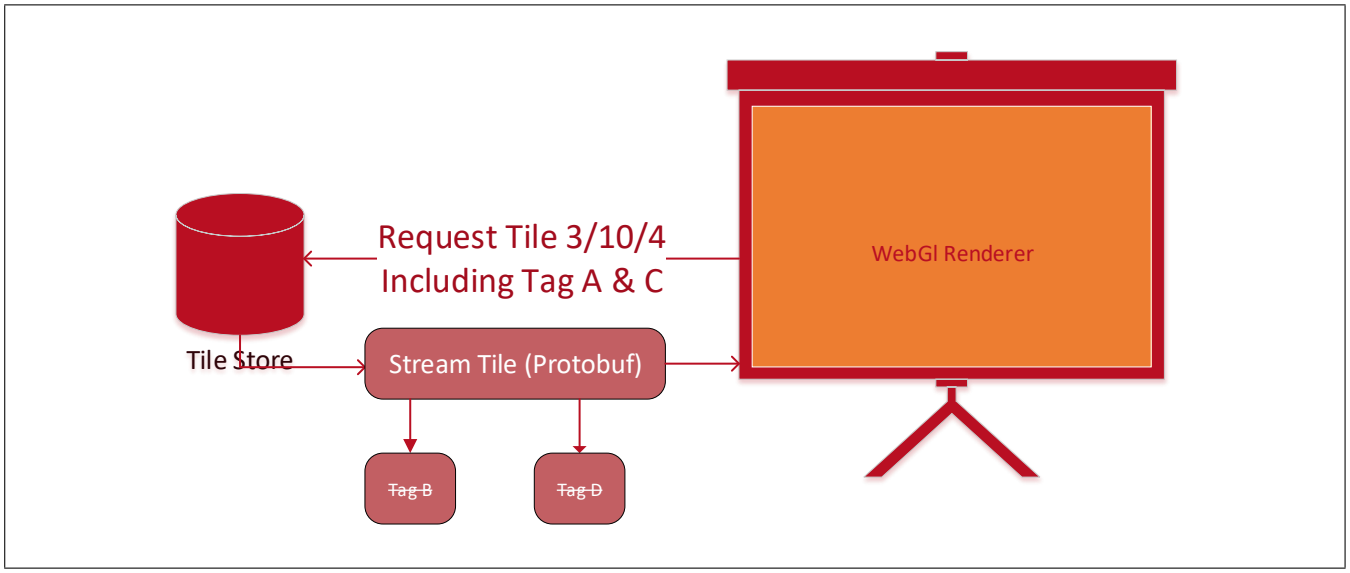


Figure 19: Modify tiles while serving them to the frontend application

9.7 Caching

Caching is always a topic on transmitting data through networks. In this workflow are two different types of caching to differentiate. On backend side tiles are either completely pre-processed to be transmitted or have to be created or modified. Their caching could optimize the performance.

On the web application side caching is done already, but in a way not suitable for every used case. Browsers can not handle caching of javascript object by themselfs, so this has to be implemented by hand.

9.7.1 Frontend

Caching in traditional map applications is done by browsers completely. The raster images are loaded in to the browsers DOM. When removed from DOM its up to the browser wheather to remove the image from memory.

Vector tiles are hold in the javascript heap. Once no one references to one tile it can be released. Currently MapBox and OpenLayers both are caching a fixed number of tiles. This is okay when dealing with small tiles as they are for example for the openstreetmap dataset. When attaching more data to the geometries or even put a thousand geoemtries into one tile one has to consider the tile size as well. An idea is to replace the LRU tile cache with a cache limited with heap size. When its exceeded the next least recent tile is removed from cache.

Also not actually visible tiles can be encoded and compressed. This would dramatically decrease the memory usage, which was one of the biggest issues in this approach.

9.7.2 Backend

On backend side no caching is implemented or tested for now. Tiles often fetched from the database could be cached in the current approach.

When talking about a more dynamic solution as on-the-fly tile creation or tile manipulation while streaming backend caching will become very important. Caching could be done on feature or even on geometry level already. This highly depends on the used data structure.

9.8 Dynamic Tile Resolution

Currently all Vector Tiles created are transformed to the same extend, which is 4096 by default. One simple optimization is to lower the extend also described as tile resolution on lower levels of detail (e.g. zoom level 0).

One has to evaluate how the tile creating process has changed. Performance is one thing, but lowering the resolution could lead in skipping of very small geometries. In some used cases this could eliminate all features on low resolutions. This is caused by transforming all geometry coordinates into integer values. Coordinates of one very small

geometry could then be all in one point. The tile creating process then has to make sure this geometry is created as a point included in the vector tile.

Styling and rendering then has to be adapted to be suitable also for points, not only for polygons.

Appendix A: Code Snippets

Listing 2: Clipping a set of features written in Kotlin

```kotlin
/**
  Clips a set of Features using given parameters
  @param features a list of features
  @param scale representing the level of detail
  @param k1 first clipping point
  @param k2 second clipping point
  @param axis the direction to clip (x or y axis)
  @return a list of features already clipped
*/
fun clip(features: List<Feature>, scale: Double, k1: Double, k2: Double, axis: Int) : List<Feature> =
  features.filter { f ->
      val scaleK2 = k2 / scale
      val scaleK1 = k1 / scale
      val min = f.min[axis]
      val max = f.max[axis]
      //condition for trivia reject
      !(min > scaleK2 || max < scaleK1)
      //condition for trivial accept center point
      // val mm = min + ((max - min) / 2)
      // (mm in scaleK1..scaleK2)
  }.flatMap { f ->
      val scaleK2 = k2 / scale
      val scaleK1 = k1 / scale
      val min = f.min[axis]
      val max = f.max[axis]
      if (f.geom.coords.isEmpty()) {
          listOf()
      } else if (min >= scaleK1 && max <= scaleK2) { //condition for trivia accept
          listOf(f)
      } else {
          listOf(Feature(
                  clipGeometry(f.geom, k1 / scale, k2 / scale, axis),
                  f.properties,
                  f.min,
                  f.max
          ))
      }
  }
```

Listing 3: Clipping one geometry written in Kotlin

```kotlin
    fun clipGeometry(g: Geom, k1: Double, k2: Double, axis: Int): Geom {
      val slice = mutableListOf<List<Double>>()
      end@for(i in g.coords.indices) {
          if (i >= g.coords.size - 1) {
              break@end
          }
          if (g.coords[i][axis] < k1) {
              if (g.coords[i+1][axis] > k2) {
                  slice.addAll(listOf(
                    intersect(g.coords[i], g.coords[i + 1], k1, axis),
                    intersect(g.coords[i], g.coords[i + 1], k2, axis)
                    ))
                  // ---|-----|-->
              } else if (g.coords[i+1][axis] >= k1) {
                  slice.add(intersect(g.coords[i], g.coords[i + 1], k1, axis))
                  // ---|-->  |
              }

          } else if (g.coords[i][axis] > k2) {
              if (g.coords[i+1][axis] < k1) {
                  slice.addAll(listOf(
                    intersect(g.coords[i], g.coords[i + 1], k2, axis),
                    intersect(g.coords[i], g.coords[i + 1], k1, axis)
                    ))
                  // <--|-----|---
              } else if (g.coords[i+1][axis] <= k2) {
                  slice.add(intersect(g.coords[i], g.coords[i + 1], k2, axis))
                  // |  <--|---
              }
          } else {
              slice.add(g.coords[i])
              if (g.coords[i+1][axis] < k1) {
                  slice.add(intersect(g.coords[i], g.coords[i + 1], k1, axis))
                  // <--|---  |
              } else if (g.coords[i+1][axis] > k2) {
                  slice.add(intersect(g.coords[i], g.coords[i + 1], k2, axis))
                  // |  ---|-->
              }
              // | --> |
          }

      }

      val a = g.coords.last()
      if (a[axis] in k1..k2) slice.add(a)
      if (slice.isNotEmpty() &&
        (slice[0][0] != slice.last()[0] || slice[0][1] != slice.last()[1]) &&
        (g.type == "Polygon" || g.type == "MultiPolygon")
      ) {
          slice.add(slice[0])
      }

      if (slice.size < 4) {
          return Geom(g.type, emptyList())
      }
      return Geom(g.type, slice)
  }
```

List of Figures

List of Tables

List of Algorithms

References

[1] Vyron Antoniou, Jeremy Morley, and Mordechai (Muki) Haklay. Tiled vectors: A method for vector transmission over the web. In James D. Carswell, A. Stewart Fotheringham, and Gavin McArdle, editors, *Web and Wireless Geographical Information Systems*, pages 56–71, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[2] J. D. Blower. Gis in the cloud: Implementing a web map service on google app engine. In *Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Application*, COM.Geo '10, pages 34:1–34:4, New York, NY, USA, 2010. ACM.

[3] Oskar Eriksson and Emil Rydkvist. An in-depth analysis of dynamically rendered vector-based maps with webgl using mapbox gl js. Master's thesis, LinkÃűping UniversityLinkÃűping University, Software and Systems, Faculty of Science & Engineering, 2015.

[4] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[5] Jianhua Feng and Jinhong Li. Google protocol buffers research and application in online game. In *IEEE Conference Anthology*, pages 1–4, Jan 2013.

[6] Petr Horak, Karel Charvat, and Martin Vlk. *Web Tools for Geospatial Data Management*, pages 793–800. Springer US, Boston, MA, 2010.

[7] Jens Ingensand, Marion Nappez, Cédric Moullet, Loïc Gasser, Olivier Ertz, and Sarah Composto. Implementation of tiled vector services: A case study.

[8] Michel Krämer. *A microservice architecture for the processing of large geospatial data in the Cloud*. PhD thesis, Technische Universität, Darmstadt, 2018.

[9] D Langfeld, R Kunze, and O Vornberger. Svg web mapping. four-dimensional visualization of time-and geobased data, 2008.

[10] Z. Liu, M. E. Pierce, G. C. Fox, and N. Devadasan. Implementing a caching and tiling map server: a web 2.0 case study. In *2007 International Symposium on Collaborative Technologies and Systems*, pages 247–256, May 2007.

[11] Jakob Nielsen. User interface directions for the web. *Communications of the ACM*, 42(1):65–72, 1999.

[12] A Olasz, B Nguyen Thai, and D Kristóf. A new initiative for tiling, stitching and processing geospatial big data in distributed computing environments. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 3:111, 2016.

[13] Linda van den Brink, Payam Barnaghi, Jeremy Tandy, Ghislain Atemezing, Rob Atkinson, Byron Cochrane, Yasmin Fathy, Raúl Garcia Castro, Armin Haller, Andreas Harth, et al. Best practices for publishing, retrieving, and using spatial data on the web, 2017.

[14] Claudia Vitolo, Yehia Elkhatib, Dominik Reusser, Christopher J.A. Macleod, and Wouter Buytaert. Web technologies for environmental big data. *Environmental Modelling & Software*, 63:185 – 198, 2015.

[15] Bisheng Yang and Qingquan Li. Efficient compression of vector data map based on a clustering model. *Geo-spatial Information Science*, 12(1):13–17, 2009.

[16] P. Yue and L. Jiang. Biggis: How big data can shape next-generation gis. In *2014 The Third International Conference on Agro-Geoinformatics*, pages 1–6, Aug 2014.